SRI International

A002/Code W; A003/Documenting; A004/Final Report • February 23, 1994

PROOF OF CONCEPT FOR THE REWRITE RULE MACHINE: INTERENSEMBLE STUDIES

José Meseguer, Principal Scientist
Computer Science Laboratory

SRI Project ECU 1505

Prepared for:
Chief of Naval Research
800 North Quincy Street
Arlington, VA 22217-5660
Attn: Dr. Keith Bromley, Scientific Officer
cc: Mary Ann Cook, Contracting Officer
Director, Naval Research Laboratory, Attn: Code 2627
Administrative Contracting Officer
Defense Technical Information Center

Contract No. N00014-90-C-0210

Approved:
Mark Moriconi, Director
Computer Science Laboratory

Donald Nielson, Vice President
Computing and Engineering Sciences Division

[illegible] • Menlo Park, CA 94025-3493 • (415) 326-6200 • FAX: (415) 326-5512 • Telex: 334486

Final Report for the Project "Proof of Concept for the Rewrite Rule Machine: Interensemble Studies"

Patrick Lincoln, José Meseguer, Babak Taheri, and Timothy Winkler

SRI International, Menlo Park, CA 94025

1 Introduction

Under the direction of Dr. José Meseguer, the Rewrite Rule Machine (RRM) team began this project on 15 August 1990, and completed the work on 14 August 1991. In addition to the project participants listed as authors, Prof. Joseph Goguen of Oxford University served as a consultant.

The main goal was to learn through simulation about the functionality and performance on realistic applications of an RRM system consisting of a collection of RRM ensemble chips (each such chip being a SIMD processor) connected on a network, and to design mechanisms to support the simultaneous parallel computation of applications across many such ensemble chips.

To achieve these goals we first built a high-level interensemble simulator and ran a collection of benchmarks on it under varying assumptions about several architectural parameters to obtain a first estimate of the communication requirements for the RRM and to determine the feasibility of those requirements in view of existing network technology. Using a second, very detailed register-transfer level simulator of a single ensemble and the performance results of a collection of applications run on it, together with modeling above the ensemble level, we also estimated interensemble performance; in this way we were able to obtain more detailed and accurate interensemble performance estimates. Mechanisms supporting parallel computations across many ensembles were also studied and designed.

Section 2 of this report gives an introductory overview of the RRM. Section 3 describes the new RRM architecture on which the detailed ensemble simulator and

the modeling of higher levels were based. Section 4 describes the interensemble computation mechanisms. Section 5 discusses simulation and performance estimation: interensemble simulations are discussed in Section 5.1, communication requirements in Section 5.2, and ensemble simulations and performance modeling of higher RRM levels based on them in Section 5.3. The code of the interensemble simulator and of several benchmarks run on it are given in Appendix A. A description of the ensemble simulator is given in Appendix B; and its code and that of benchmarks run on it are given in Appendix C.

2 Overview of the RRM

Following an overview of the Rewrite Rule Machine (RRM) architecture and model of computation with special emphasis on the new ensemble design, we discuss performance estimates based on simulation. The architecture is a multilevel hierarchy, which is SIMD at the lower (chip) levels, and MIMD at the higher levels. This enables the RRM to combine the advantages of the SIMD and MIMD approaches. The RRM model of computation is concurrent graph rewriting, which supports extremely fine-grain parallelism, dynamic resource allocation, and simple semantics.

Since performance estimation for a machine like the RRM is difficult, we must carefully justify our approach. We discuss the problems and how we address them later in this document. Our approach to performance estimation may be summarized as follows: we chose a diversity of problems to stress the design in different ways, including communication, memory, and computation; we chose problems representative of different application areas; and we built and used different simulators to get a variety of performance estimates.

2.1 Multigrain Concurrency and Applications

Many important real-life applications involve a number of diverse, relatively independent processes, many of which are computationally homogeneous. For example, a large simulation problem may involve many independent, loosely coupled processes.

Let us call a computation *homogeneous*, if at each moment it consists of many instances of the same instruction being applied to many data items in parallel; sometimes this is called *data parallelism*. While many familiar numerical algorithms have this form, many complex computational tasks are *locally homogeneous* but *globally inhomogeneous*.

Because of its very fine-grain SIMD parallelism at the chip level combined with its flexible coarser-grain MIMD parallelism at the network level that allows different chips to work on very different subtasks of the same problem at once, the RRM can

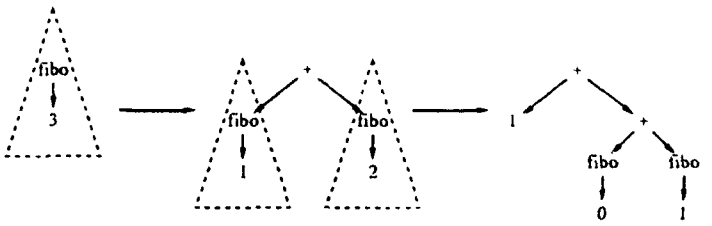


Figure 1: Concurrent Rewriting of Fibonacci Expressions

exploit a problem's parallelism at several levels. We call this property *multigrain concurrency*; it makes the RRM very well suited for solving not only homogeneous problems, but also complex, locally homogeneous but globally inhomogeneous problems in many areas, including discrete event simulation, decision support systems, rapid prototyping, vision, computational geometry, automated deduction, finite element methods, neural nets, and hardware simulation.

2.2 Combining SIMD and MIMD

At present, the two main approaches to massive parallelism are SIMD machines and MIMD multicomputers. Examples of the state of the art in each category are the Connection Machine, CM-2 (Thinking Machines Inc. [14, 4]), and the MP1216 (MasPar Computer Corporation [23]), for SIMD computers; and Mosaic (Chuck Seitz, Caltech [22]), the J-machine (William Dally, MIT [5]), Paragon (Intel Corporation), and the CM-5 (Thinking Machines, which simulates SIMD by MIMD broadcast), for MIMD computers. These two approaches are quite different. Each has unique advantages not shared by the other approach. The strength of SIMD machines is their exploitation of fine-grain data parallelism, which makes them a good choice for homogeneous problems; their weakness is their centralized control, executing the same code everywhere, which makes them perform poorly on large nonhomogeneous applications. MIMD machines are much more flexible because they allow different code to be run in different processors simultaneously; however, their communication—typically asynchronous interprocessor message passing over a network—is not well suited to data parallelism.

A key goal of the RRM is to combine the best of these two approaches in a single architectural design. It shares with SIMD machines the capability for fine-grain data parallelism, which is carried to an even finer level in the RRM ensemble; however, because of its decentralized MIMD control, the RRM can perform well on both homogeneous and nonhomogeneous problems, whereas SIMD machines can excel only on homogeneous problems. Compared with MIMD machines, the RRM enjoys the same flexibility and generality, based on distributed control and asynchronous

message passing, but because the RRM is SIMD at the chip level, it can exploit fine-grain data parallelism locally, even for highly nonhomogeneous applications, whereas, at present, purely MIMD machines can get large degrees of parallelism only at the interprocessor level.

2.3 Programmability

The RRM is programmable in a wide variety of declarative ultra-high-level languages that permit massive exploitation of implicit parallelism and ease the creating and porting of parallel programs. We believe that declarative languages are good choices for programming such applications as vision, real-time plant control, simulations, and expert systems, because they do not require explicit commitment to specific forms of synchronization or scheduling. These convictions are supported by extensive simulations, and by compilation techniques [12, 1, 20] making functional (e.g., OBJ [8]), object-oriented (e.g., Maude [17], FOOPS [11]), and relational (e.g., Eqlog [10]) programming languages easy to compile into RRM code.

However, it is a fact of life that some parts of large applications programs have already been written, and it may not be practical to rewrite them in a declarative language. Because its flexible model of computation also supports imperative features, a compiler for the RRM from a conventional language, even a sequential one, could be written relatively straightforwardly.

2.4 The Concurrent Rewriting Model of Computation

The RRM's model of computation is **concurrent rewriting**. In this model, data are **terms** constructed from a given set of constant and function symbols, and a program is a set of equations that are interpreted as left to right **rewrite rules**. The lefthand side (abbreviated LHS) and righthand side (RHS) of a rewrite rule may have variables as well as function symbols. A variable can be instantiated with any term of the appropriate sort, and a set of instantiations for variables is called a **substitution**.

A rewriting computation starts with a given term as its **data** and a given set of rewrite rules as its **program**. Applying a rewrite rule has two phases, called **matching** and **replacement**. The matching phase attempts to find a substitution that yields a subterm of the input term when applied to the rewrite rule's lefthand side. Then, in the replacement phase, the matched subterm, called the **redex**, is replaced by the righthand side of the rule, instantiated with the same substitution. Rules are applied until no more matches can be found; then the resulting term is called **reduced** and considered to be the final result.

In the concurrent rewriting model of computation, more than one rule can be applied at once, and each rule can be applied to many subterms of the given term

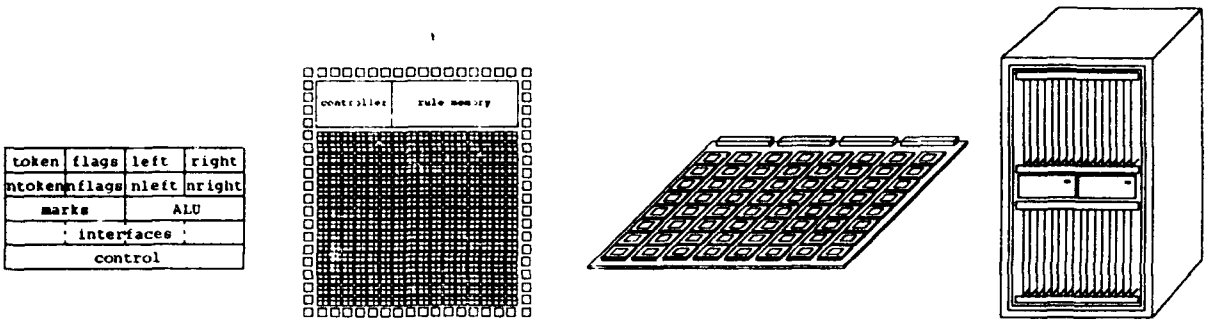


Figure 2: Hierarchical Structure of the RRM

at once. Let us explain this by example. Here is a simple program to compute the Fibonacci numbers:

```
(1)   fibo(0) = 0
(2)   fibo(1) = 1
(3)   fibo(N) = fibo(N-2) + fibo(N-1)
                  if N > 1
```

If you give `fibo(3)` as data, the top node will match rule (3); thus the whole term will be replaced by

```
fibo(1) + fibo(2)
```

In the next step, the first `fibo` node will match rule (2), and the second `fibo` will match rule (3) again, and the simultaneous application of these rules yields

```
1 + (fibo(0) + fibo(1))
```

in just one step of concurrent rewriting. Figure 1 illustrates these two concurrent rewriting steps, using tree representation for expressions.

We say that a concurrent rewriting computation is **SIMD**, when just one rewrite rule is applied concurrently at each moment; in the RRM, this style of concurrent rewriting is realized by an **ensemble** chip, as explained later. If several rules are concurrently being applied, each to possibly many instances, we have **MIMD** concurrent rewriting; this general case is the correct model for the RRM as a whole. See [9] for general background on the concurrent rewriting model, [6] for definitions of SIMD and MIMD rewriting (called *parallel* and *concurrent* rewriting in that paper), and [18, 19, 17] for a definition of concurrent rewriting as deduction in rewriting logic and a systematic treatment of concurrent object-oriented computation by means of concurrent rewriting.

Two additional topics treated in [9] deserve mention. The first is **sharing**, which permits a common substructure of two or more given structures to be shared between

them, rather than requiring that it be duplicated. This leads to directed acyclic graphs rather than just trees. The second topic is **evaluation strategies**, which are annotations that impose restrictions on concurrent execution in order to improve the performance of parallel computations. A **strategy** for a function f of n arguments consists of a set $\{i_1, ..., i_k\} \subseteq \{1, ..., n\}$ indicating the argument places that should be already reduced before a rewrite rule match for f is attempted. For example, `if_then_else_fi` is typically computed with strategy $\{1\}$, and integer addition with "bottom-up" strategy $\{1, 2\}$.

3 The RRM Architecture

The RRM architecture is **hierarchical**, with each unit consisting of a collection of cooperating units at the next lower level. The most basic processing element is the **cell**, with four cells making up a **tile**. An **ensemble** chip contains hundreds of cells (576 is our current estimate). A **cluster** is a collection of ensemble chips connected on a board, and the machine as a whole is a **network**. Figure 2 provides a pictorial representation of the RRM hierarchy.

A single ensemble yields very fast, extremely fine-grain SIMD rewriting, but RRM execution is coarse-grain MIMD at the cluster and network levels, since each ensemble independently executes its own rewrites on its own data, communicating with other ensembles when necessary.

3.1 Cell, Tile, and Ensemble Architecture

The most basic computational element in the RRM is the **cell** [16, 2], which stores one data item with pointers to other cells, and also provides basic computational and communication capabilities; thus cells mix storage, computation, and communication. A cell consists of:

- Several **registers** (mostly 16-bit), including:
  - `token`, which encodes the operation or constant symbol of a data node,
  - `left` and `right`, which point to the descendant nodes[1],
  - a 32-bit `marks` register, which holds volatile information (similar to condition codes),
  - `flags`, which holds less volatile information, such as type and reduction status,

[1] Unary operations only use `left`, and n-ary operations for $n > 2$ are decomposed into binary ones.

- Twelve general-purpose registers, including `ntoken`, `nleft`, `nright` and `nflags`.

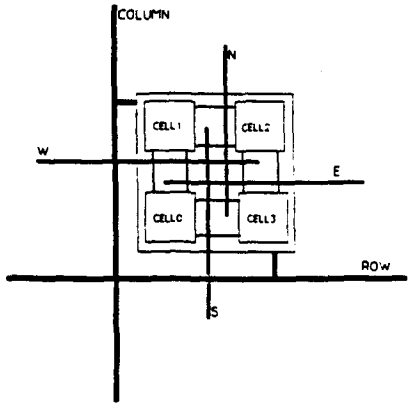
- An **ALU** to operate on and test the contents of registers.

- **Interfaces** to communication channels and the controller.

We divide the silicon area of the ensemble chip into a 12 × 12 mesh of **tiles**, each with four cells. Adjacent tiles are directly connected by short wires, so that placing logically linked nodes in cells located in adjacent tiles permits very efficient communication. Placing several cells in one tile increases the probability of logically related data being in adjacent cells.

Our new ensemble design is simpler and has substantially better overall performance than previous designs [7, 2]. Its simpler instructions allow a faster clock (100 MHz seems a reasonable estimate) and provide much better support for communication between cells.

An ensemble has a single SIMD **controller** that broadcasts its instructions to all cells. The controller can obtain very fast feedback (one clock cycle) about the state of the cells (such as type of data and operation symbols in cells, remote references, success or failure of an instruction, and termination) and can use such feedback to branch to different SIMD code segments. Obeying SIMD instructions, cells can communicate with adjacent cells (each cell has 16 adjacent cells in its 4 adjacent tiles) to find local patterns for rewriting; hundreds of such patterns may be found and transformed simultaneously. Other SIMD instructions allow communication among nonadjacent cells using special row and column buses, relocation of data, and input-output. The short buses, called *ports*, allowing fast communication of each cell with the 16 cells in the north, south, east, and west (N, S, E, and W) neighboring tiles, as well as the row and column buses used for communication of nonadjacent cells under SIMD control, are shown in the figure below.



SIMD concurrent rewriting takes place by broadcasting instructions that implement matching and then replacement of the patterns found. Although for very regular computations it is possible to avoid remote—i.e., not physically adjacent—references within a single ensemble, in general the dynamic nature of the computation will require remote references, and then matching will require *relocation* of some data. This is accomplished with specialized instructions and chip-level hardware support, including cell and tile features, and buses for communication between distant cells.

We use a reference counting scheme for storage management, both within ensembles and in the RRM as a whole. We have fully simulated the details of this within the ensemble for the examples discussed later in this report.

3.2 Cluster and Network Architecture

The cluster architectural level corresponds to board-level structure in the actual implementation. At this level, ensemble chips can be arranged in a two-dimensional (2D) mesh with fast connections to each of four neighbors, giving 8 connections per ensemble (4 in and 4 out). With current technology, these could be 16-bit-wide connections running at 50 MHz, giving 800 Mbps per connection and 6.4 Gbps total bandwidth per chip. Additional interconnection hardware at the board level beyond the fast, local connections is also desirable, as in the iWARP [3] and DataWave [21] designs. The performance we assume is not that much beyond that provided by these designs; the iWARP has 8 ports, each 8 bits wide at 40 MHz, giving 320 Mbps per port and 2.56 Gbps total (100 to 150 ns latency), and the DataWave has 8 ports, each 12 bits, at 60 MHz, giving 5.76 Gbps total. We are estimating that a cluster will have about 100 ensembles.

The network level interconnection for the RRM has not been fixed. We have been considering the wormhole routing networks of Seitz [22] and Dally [5]. Actual realizations of these designs have achieved high communication rates: 205 Mbps for Ametek 2010, and 200 Mbps for the Intel Paragon. For a 2D mesh, average case communication time for 10,000 nodes is estimated at 1885 ns, or 188 clock cycles. For a 3D mesh, the average case communication cost for 10,000 nodes is estimated at 976 ns, or 98 clock cycles.

In general, interchip communication in the RRM is *asynchronous* message passing that imposes no critical timing requirements on the network or switching technology. Thus, the RRM can exploit the best communication technology available, and take advantage of any future improvements. However, the RRM can exploit locality and use fast local interensemble connections at the cluster level to get very high performance for certain problems.

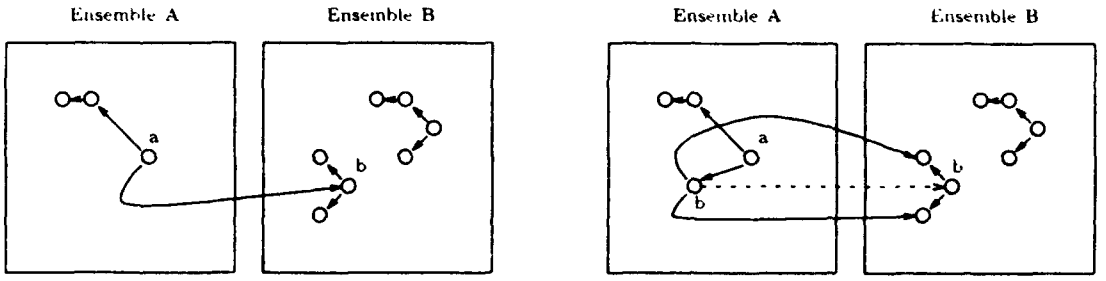


Figure 3: Before and After Creation of Ghost (of **b**)

4 Interensemble Computation

Sometimes active cells in one ensemble need information from descendants in another ensemble. We call references from one ensemble to another ensemble *distant* references, to distinguish them from the *remote* references that occur from cell to nonneighbor cell within a single ensemble. Although distant references can be reduced by relocating data to ensembles that reference it most often, it is impossible to completely eliminate distant data references, even using static memory allocation, because, in general, structures will not fit in a single ensemble. To efficiently support interensemble communication, we have developed two related mechanisms.

For symbolic computation, where data is laid out dynamically and computation is asynchronous or delay-insensitive, we use an incremental symbolic cache approach. When a distant reference is made, and it is determined that the distant node should not be relocated to the local ensemble, then a *ghost node* is instead allocated in a cell of the local ensemble, and data from the target of the distant reference is copied into the ghost node. However, unlike true relocation, the ghost node is *prevented from being the root of a rewrite*, i.e., is temporarily frozen. Also, a ghost node maintains a copy of the original distant pointer, and thus acts as a passive incremental "symbolic cache" of data that actually resides on another ensemble. After some time, under SIMD control, ghost nodes flush their data and use the stored distant pointer to refresh their contents. This flush-refresh of ghost information may be performed at any time. In addition, at some times the parent of a ghost may copy the distant pointer from its descendant ghost, and then cause deletion of the ghost.

For example, in Figure 3, in the before (left) picture, ensemble **A** contains a cell labeled **a** that has a distant pointer to a cell labeled **b** in ensemble **B**. In the course of pattern matching, cell **a** requires information from its descendant **b**. In the after (right) picture, a ghost node for **b** has been created in ensemble **A**, and the distant pointer from **a** to **b** has been replaced with a local pointer from **a** to the new ghost of **b**. Thus the ghost of node **b** has distant pointers to the children of **b**, and also has a

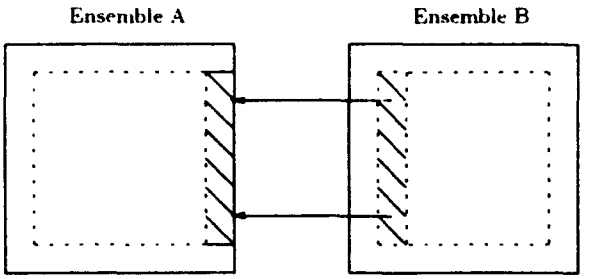


Figure 4: Systoli. Interensemble Computation

copy of the original distant pointer (shown as a dashed arrow to node **b** in ensemble **B**). Note that this process cannot continue indefinitely, since ghosts are not allowed to initiate the matching process themselves. Thus even if the structure underneath **b** is large, only that portion of the structure needed to verify a match rooted at **a** is ever copied to ensemble **A**.

The mechanism used in the systolic case is similar in spirit to the symbolic case described above, but can be implemented somewhat more efficiently, due to the locality of reference that (in part) characterizes systolic computations. Because this locality does not change during a computation, we should place elements that communicate frequently on the same ensemble. As in the symbolic case, structures may be too large to fit on a single ensemble, and then we must place portions of the problem on neighboring ensembles, while keeping local copies of the border data current on both ensembles. Since systolic computation is synchronous and delay-sensitive, we must ensure that the border data is updated correctly when it is read by the local ensemble. In general the systolic computation must wait every cycle for the block transfer of data between ensembles.

In Figure 4, ensembles **A** and **B** each contain an area of active cells delineated by the dashed box. Outside this box are border cells that do not necessarily perform computations, but instead store copies of the near-edge cells of neighboring ensembles. Figure 4 shows a block copy of information from active cells in ensemble **B** to (passive) edge cells in ensemble **A**. After information from each neighboring ensemble is copied into ensemble **A**, the next step of computation can proceed.

In many cases we can overlap communication with computation. This potential overlap, or rudimentary pipelining of I/O and computation, is another consequence of our architectural choice of multiple cells per tile. The current design of the ensemble with four cells per tile allows simultaneous systolic computation of four distinct two-dimensional layers at a time. In fact, one or two layers could perform I/O at the same time that the other layers perform their systolic computations. In this way, we may hide some of the potential I/O penalty of interensemble computations.

4.1 Load Balancing

Allocation in an ensemble normally ensures that allocated cells are neighbors of the allocating cell. However, when an ensemble becomes too full, allocations are made on other ensembles. This process can be described as *pushing out* computational subtasks. The SIMD controller can gather (perhaps imprecise) information about the utilization level of an ensemble in order to determine when the ensemble is full. For certain computations, it may be advisable to push out subtasks at the outset. Large symbolic computations usually require building and manipulating very large term structures, which may be distributed over several ensembles when they are initialized, may be distributed explicitly by a specially tuned SIMD broadcast, or may migrate implicitly to neighboring ensembles during computation.

Allocation is important in architectures like the RRM, due to the sensitivity of computation to locality. Thus, initial placement may have a large impact on performance, especially for relatively short computations with large amounts of data. After initial allocation, the compiled SIMD code may explicitly push subcomputations out of an ensemble, perhaps forming a ghost node in its place. Thus the local copy does not perform rewrites itself, although it would still participate passively in other rewrites.

Finally, automatic migration can be performed by pushing subtasks out of an ensemble based on the depth of the subterm from a root node of the ensemble, forcing subcomputations to be pushed out more quickly. However, spreading computation more quickly, and thus more evenly among ensembles, trades off against interensemble communication overhead. The techniques for interensemble computation already described above substantially alleviate this overhead, but it still exists.

5 Simulation and Performance Estimation

Estimating the performance of computer systems is a difficult art at best, and is even more difficult for radically new machines that have not yet been built. The performance limitations of simulators mean that large problems are very difficult to run. Testing different aspects of a design on the largest possible problems may force using multiple simulators to abstract different details for various choices of performance measure and problem. But then it may be difficult to justify the abstractions, and to ensure that the problems fit the assumptions behind their justifications. For the RRM, these difficulties seem particularly acute, because of the high performance figures that we seek to justify.

Given the serious performance limitations posed by trying to simulate our architecture on the workstations available at the time our research was carried out (limitations still applying today to a good extent) the approach to performance estimation that

we adopted was a hybrid one:

- *Interensemble simulations*, in order to be computationally feasible, were performed at a high level of modeling using a high-level *interensemble simulator* (see Section 5.1) not detailed enough to yield precise quantitative information, but useful for obtaining preliminary estimates of communication requirements (see Section 5.2) and also for gaining experience with interensemble computations, and for getting some rough performance estimates.

- *Ensemble simulations* for our new RRM ensemble design were performed using a very detailed register-transfer level *ensemble simulator* (see Section 5.3). By running a widely varied collection of applications on this simulator, very precise estimates were obtained at the ensemble level. Using modeling—based on architectural assumptions consistent with the interensemble simulation experiments and feasible with current technology—for the higher levels of the RRM architecture, we were then able to obtain more detailed interensemble performance estimates for the RRM as a whole.

5.1 Interensemble Simulations

We developed an interensemble simulator for the RRM and used it to develop and test ideas about interensemble communication mechanisms and strategies; the code of this simulator as well as that of the benchmarks run on it is given in Appendix A. For this simulator, the RRM as a whole was modeled as a 2D array of clusters which, themselves, were 2D arrays of cells. Clusters were assumed to be interconnected in such a way that one could view the machine as a whole as a 2D array of ensembles. We introduced the notion of clusters to model a difference in technology between board-level interconnection and interconnection at a larger scale. The overall 2D topology was chosen because it was a relatively modest interconnection structure that should be realizable in practice. The simulator was instrumented to keep track of communication at different levels so that we could get some estimates of the communication requirements of the RRM as a whole and between clusters.

The simulator was a very high level simulator, manipulating term structures by applying rewrite rules, but a term was considered to be located in a particular ensemble and ensembles had some limitations on the total size of the terms they could contain. The ensembles apply rewrite rules independently, and then apply strategies to determine if terms should be pushed out, making more room, or pulled in, making ghosts. The basic high-level actions are those of pushing out a subcomputation (somewhat similar to a remote procedure call) and copying results or partial results back in (which returns a final result or provides a cached version of a partial result).

Two primary strategies were used for deciding when to push subterms out of an ensemble. One strategy was based on a threshold value for the depth of a term below a root in the ensemble. If this depth threshold was chosen to be fairly shallow, then the term would be forced to rapidly spread out through the RRM. This would have the benefit that a given ensemble would be unlikely to contain only an upper part of the tree that was waiting on subcomputations to complete and so be idle. This strategy would be applied regardless of how full the ensemble was; the size limitation would mainly come about when an ensemble refused to accept a subcomputation being pushed out from elsewhere. Such a strategy is analogous to the treatment of allocation within an ensemble.

The other strategy used a fullness threshold to decide when to push out a subcomputation: if an ensemble becomes very full, a major subcomputation is pushed out. The strategy for selecting the subterm to be pushed out is to select either whole rooted terms if they are small or to select a top-most large subterm of a very large rooted term. (In practice, it is necessary to have a second fullness threshold that determines when the ensemble will accept pushed out subcomputations from other ensembles.)

Some of the problems simulated were: numeric Fibonacci number calculation, Peano arithmetic Fibonacci, bubble sort, merge sort, and matrix multiply. These examples were chosen because the patterns of computation are quite different in these examples and seem to represent an interesting variety. The amount of time required to do the simulations was a limiting factor on the size of problems run for these examples.

Different versions of the simulator were used in simulations. Changes were introduced in order to make the simulations more realistic; for example, limits on the amount of communication between ensembles in a cluster or between clusters were imposed in some versions, to test the impact of such limits.

Some of the parameters that we experimented with were: time penalties for both intra- and intercluster communication, size of ensemble for allocation, intra- and intercluster communication limits, relocation size threshold (an ensemble must be below a given limit of occupancy before it can relocate structures inward), push-out threshold (and an associated goal for the size of the ensemble when subcomputations have been pushed out), and a value controlling when subcomputations are pushed out based on the depth of a term below a root in an ensemble. Most of these parameters were not critical in that small variations in their values had only small effects on the performance of the simulated RRM. The robustness of the performance relative to these variations is a very positive result. For the simulations performed to generate estimates (discussed later), values were chosen that were as realistic as possible and on the conservative side (i.e., values that would tend to produce the least favorable result).

The high level of abstraction of the simulation process means that the results cannot be expected to be precise predictions of the behavior of the RRM. On the other hand, we expect that the large-scale behavior should be roughly similar. We also now believe that the development of multiprocessor interconnection technology is proceeding so rapidly that we can expect to have networks that are very well adapted to the RRM architecture.

5.2 Communication Requirements and Networks

The following subsections give an indication of the current state of the art in high-performance, low-latency interconnection networks and then present the specific estimates of the communication requirements derived from the interensemble simulations.

5.2.1 Prospects for High-Performance Interconnects

Demand for very high performance interconnects is being driven both by tightly coupled shared memory systems and by more experimental distributed memory systems. Theoretical results have been abundant in this area, from theoretical studies done for phone systems to a large literature on hypercubes and their variants. However, careful comparisons and evaluations of tradeoffs and actual practical engineering experience are just being developed for the kind of large-scale interconnects that interest us. Just constructing an interconnect for 500 processors is a major project, probably too large for the academic context and hard to justify either commercially or in government funded research without a clear use (i.e., an overall system architecture using the interconnect).

The analysis of design tradeoffs in the thesis of Dally has led to a whole new generation of wormhole routing interconnects used in machines such as the Ametek 2010, the Fujitsu AP1000, and the Intel iWARP [3]. The iWARP processor is intended for very high performance, very fine grain systolic computation. A new processor with a similar goal is the ITT DataWave [21]. There are also next generation designs such as Seitz's CalTech MOSAIC project and Dally's J-machine project at MIT. These designs achieve impressive communication rates: 205 Mbps for Ametek 2010, 200 Mbps for Fujitsu AP1000, 160 Mbps per port and 2.56 Gbps aggregate for the iWARP (8 ports, each 8 bits, at 40 Mhz and with 100 to 150 ns latency), and 6 Gbps for the DataWave (8 ports, each 12 bits, at 60 Mhz). Both the iWARP and DataWave examples are interesting because systolic computation is even more demanding of the interconnect than the RRM design is expected to be. Progress can be expected to continue along these lines, and good designs will be developed and tested for systems with large numbers of processors (500 to 1000 or more). For example, the MOSAIC system is planned to have 16000 processors and will be based on a 3D mesh

$(32 \times 32 \times 16)$.

Another feature of the iWARP and DataWave designs is that they are single-chip processor designs. This allows the 8×8 prototype for iWARP to fit on a single board. With current technology, this gives a significant advantage as wiring densities can be much denser on a board than off. The other machines (Ametek 2010, Fujitsu AP1000) were based on processing nodes with from one to four processors per board. The goal for the RRM is to have the complete ensemble consist of an ensemble chip (each with 576 processing elements (PE's)) perhaps with one or two more chips for extra memory and routing, thus also allowing many processors per board. We believe that low latency and local high performance may be much more important than global bandwidth.

In the future, it is likely that there will be very important solutions based on mixtures of technologies. For example, CMOS to GaAs with silicon lasers and a optical interconnect could provide multi-gigahertz rates over a single optical fiber. To reduce the number of wires and maintain bandwidth, it is necessary to increase the frequency or rate of operation, which suggests a change in basic technology. However, it is likely that CMOS, or related technologies will continue to provide the highest density available, which suggests that a mixed approach might be desirable. The important point here is that there is much room for improvement of interconnection technology and that the RRM —because of its asynchronous model of computation— can easily take advantage of such improvements.

5.2.2 Communication Requirements

We have used the high-level interensemble simulator on a suite of characteristic examples to estimate upper bounds on the communication demands of an ensemble. Our upper bounds seem to be in the range of the newer network and interconnection technologies such as the wormhole routing networks of Seitz [22] and Dally [5].

As mentioned, interchip communication in the RRM is *asynchronous* message passing communication and imposes no critical timing requirements on the network or switch technology. This makes the RRM capable of exploiting the best communications technology available, and of taking advantage of any future improvements in such technology. The following are important observations on the communication requirements of an ensemble:

- Estimated ensemble I/O rate: 160 Mbps to 520 Mbps (estimate based on specially instrumented interensemble simulations).
- Pins are not a bottleneck: realistic current estimate is 4 Gbps (100 pins at 40 Mhz).

- Communication capacity seems to be in the range of newer network and interconnection designs (Seitz [22], Dally [5], iWARP[3], DataWave[21]).
- Average communication performance is enough; the RRM design doesn't make critical timing requirements on the communication network.

5.3 Ensemble Simulations and Interensemble Performance Modeling

The new ensemble design and estimates of its performance have been validated by running a variety of benchmarks on a new ensemble simulator written in C, which models the ensemble computation in great detail at the register transfer level. A detailed description of this simulator is given in Appendix B. The code of the simulator as well as that of several applications run on it are given in Appendix C.

Our simulations at the ensemble level have a great level of detail and give quite accurate performance estimates, but our overall performance estimates for the RRM are still preliminary, and more studies and experiments are required to increase their accuracy. The present estimates are based on detailed ensemble simulations, high-level interensemble simulations, estimates of communication requirements, and analysis using simple approximate models. More definitive performance estimates will require more detailed simulations and analytic studies for a wider collection of examples and applications.

The performance models are based on simple predictions of the computation times for specific strategies for performing the computations. We discuss RRM performance predictions for a variety of examples chosen because their patterns of computation are representative of different kinds of computations; they represent basic examples of general symbolic computations (numeric Fibonacci and the TAK function), highly regular symbolic computations (sorting), and discrete event simulations of a systolic nature (fluid flow and a simple hardware simulator).

When describing RRM performance at the cluster or network levels, we specify efficiency as a percentage of the ideal performance. The ideal performance corresponds to a linear extrapolation of a single ensemble's performance, i.e., a linear speedup. We will also give "idealized Sun-relative speedup," which simply is the product of the number of ensembles, the Sun-relative speedup, and the efficiency.

We assume a 100-MHz clock and a 12×12 array of tiles requiring approximately 6 million transistors. These figures seem achievable since speeds and sizes of this kind have already been demonstrated. For example, the 1991 Hot Chips conference [15] presented two chips with 100-MHz clocks (one of them with 4.1 million transistors), and another chip with 14 million transistors.

There are many different performance measures for machines, including machine instruction execution rates, and actual elapsed time. The most *intrinsic* ensemble performance estimate is the number of clock cycles needed for a given computation. By assuming a specific clock rate, this measure can be translated into seconds. However, some *relative comparison* of performance between the ensemble and existing sequential processors is also desirable. We use the *Sun-relative speedup* for this purpose. To obtain this comparative measure we write one program in ensemble SIMD code or with rewrite rules, and another in efficient C. By comparing the actual performance of the C program on a Sun workstation with the performance of the SIMD code on the ensemble simulator, we obtain for each problem a speedup measure called "Sun-relative speedup." In our case, we take a Sun SPARCstation IPC as the basis for comparison. This could also be used to assign a "MIPS" rating to the ensemble by multiplying this speedup by the published MIPS ratings of the specific Sun workstation, which is roughly 15 MIPS for the SPARCstation IPC. In most cases, the aim is to compare a good algorithm for a problem on the RRM with a good sequential algorithm on a Sun. In some cases, the optimized sequential Sun version involves significant variations from the algorithm used on the RRM. When we discuss each benchmark below, at the ensemble level and levels above, we mention the specific assumptions made.

5.3.1 Performance Estimates for TAK

The TAK benchmark is a subtle modification of the function Ikuo Takeuchi originated specifically to test Lisp systems. The modification accidentally introduced by Richard Gabriel and John McCarthy makes the function more difficult to optimize, but preserves its simple, recursion-intensive structure. We have implemented TAK for the RRM and in C for purposes of comparison. The Lisp and C code are shown below:

```
(defun tak (x y z)
  (if (not (< y x))
      z
      (tak (tak (1- x) y z)
           (tak (1- y) z x)
           (tak (1- z) x y))))
```

```
tak(x,y,z) register int x,y,z;
{ int r1, r2, r3;
  while (1) {
    if (x<=y) return z;
    r1 = tak(x-1,y,z);
    r2 = tak(y-1,z,x);
    r3 = tak(z-1,x,y);
    x = r1; y = r2; z = r3; }}
```

Because our most detailed simulations are limited to a single ensemble, we have used the arguments 12, 8, 4 instead of the more traditional 18, 12, 6. The RRM code completes this benchmark in 22,428 cycles, while the C version finishes in .0015 second on a SPARCstation IPC. This leads to a Sun-relative speedup of 6.7 (= .0015/.00022428). We currently don't have cluster or RRM estimates for this example.

5.3.2 Performance Estimates for Numeric Fibonacci

A strategy for computing numeric Fibonacci—which yields a simple approximate model for estimating performance—is to do the computation directly if it fits in one ensemble, and otherwise apply the last of the following rewrite rules for `fibo`

```
fibo(0) = 0
fibo(1) = 1
fibo(N) = fibo(N-1) + fibo(N-2) if N>1
```

once, and then push out the subcomputation of `fibo(N-2)`, to proceed in parallel with that of `fibo(N-1)`, which either may be done locally or may push out further subcomputations. This strategy always keeps a significant subcomputation for the current ensemble. Detailed ensemble simulations allow quite accurate estimates of time required for n up to 10 (it is linear in n). By comparing with the time required to run the same algorithm in C on a Sun workstation, we obtain a Sun-relative speedup of 6.7. The cost for larger n is the time to set up the subcomputations, plus the maximum of the cost to finish the local subcomputation and the cost to finish the pushed out subcomputation, plus the cost to finish the computation. Assuming that network I/O can be overlapped with SIMD broadcast, but that transferring a simple expression like `fibo(10)` out of an ensemble or transferring a result such as 2584 takes just a small number of SIMD instructions, the complete time to compute the numeric Fibonacci can be modeled by a recursive function allowing different assumptions about the network communication delays. For very fast networks, the network communication times and the computation times (for setup and finishing) are roughly comparable, so that network I/O cannot dominate the overall computation time (usually it will be overlapped with computation).

The cost of numeric Fibonacci within an ensemble is approximated by

$$\text{fibens}(n) = 250 \times n - 50$$

for $n \geq 3$. The approximate cost to compute the n-th Fibonacci, for $n \geq 10$, is then

$$\begin{aligned}\text{fibgen}(n) = {} & \text{simdcost}+ \\ & \max(\text{fibgen}(n-1), \\ & \quad \text{fibgen}(n-2) + \text{pushcost})\end{aligned}$$

where simdcost is the SIMD execution cost to set up the subcomputations, push out, pull in, and finish the Fibonacci computation (approximately 300 clock cycles), pushcost is the cost to do two I/O operations (estimated to be less than 200 clock cycles for 10,000 ensembles, and $\text{fibgen}(n) = \text{fibens}(n)$ for $n < 10$. With these estimates the simdcost dominates, I/O is overlapped, and efficiency is very good. For larger n, $\text{fibgen}(n) = 300 \times n - 455$. For a 10,000-ensemble RRM, the predicted worst-case efficiency for this example is 88%, which seems quite encouraging. The idealized Sun-relative speedup is 59,000.

5.3.3 Performance Estimates for Sorting

A simple way to sort a sequence of numbers on an RRM ensemble is to use a 2D exchange sort that uses both "bubblesort" exchanges of consecutive elements of the sequence and "shortcut" exchanges between nonconsecutive elements. By appropriate placement of the sequence within an ensemble, both types of exchanges can be accomplished by simple, local transformations. For a 23×23 array of values we can form a linear sequence of numbers in the array by going down the first tile column, up the second column, and so forth. We can also establish horizontal shortcut links between list elements that are adjacent elements of the same row. By folding the 2D array twice, it is possible to embed the array in an ensemble and fit a list with 23×23 ($= 529$) elements inside an ensemble in such a way that all links are direct neighbor-to-neighbor connections. The 2D exchange sort algorithm alternates bubble sort exchanges between consecutive elements in the sequence with shortcut exchanges between nonconsecutive, but horizontally adjacent elements. For a list of length n placed in this manner, the time to do a 2D sort within a single ensemble is proportional to $\sqrt{n}$, and requires approximately $221 \times \sqrt{n} - 468$ clock cycles. The average number of instructions for either the bubblesort or the shortcut exchange phases is 42, giving a main loop size of 84. Comparing with the time taken by a simple quicksort algorithm written in C and running on a Sun workstation yields a Sun-relative speedup of 127. Uniformly distributed random data was used for the tests.

At the interensemble level, one can use the same pattern, i.e., ensembles in a mesh and interchanges in the long chain or rows, but interchanges will always exchange

the maximal value from one ensemble with the minimal value from the next. For this problem, the computation within an ensemble has a structure different from the structure at the cluster level and higher. The simple, fixed connectivity is one advantage of this approach; it should be possible to allocate ensembles so that all I/O connections are local, best-case links. The data would be broken into chunks, which start interchanging data internally and across ensemble boundaries, in one of two directions, at their endpoints. When data items are exchanged across a boundary, an item is pushed out and another is pulled in, preserving the size of the chunk in the ensemble. It seems better not to have a lock-step process, in which data items are always exchanged, but instead to exchange data only when there is a need, e.g., when a new value has been interchanged into an end position.

In order to get estimates at the cluster level, a special simulator was written in C that simulates the two-level 2D sorting algorithm and calculates clock-count estimates. Note that, because of reduction of the bandwidth through a cross section of the machine, one expects that sorting at the cluster level should be at least 23 times slower than within an ensemble. Since there are 100 ensembles at the cluster level, one might still see some further speedup; however, the algorithms are more complex and less efficient. If very fast neighbor-to-neighbor connections can be used at the cluster level (and this should also be possible at the level of the RRM as a whole), then exchanging data with a neighbor should take only 5 to 10 clock cycles. The phases consist of local plus global linear exchanges, with an additional smallest to largest shortcut, and local plus global row exchanges. The additional cost, due primarily to communication, of global operations is estimated at 20 to 30 instructions. The estimated time to sort in a cluster was compared against the time to quicksort on a Sun giving an estimated Sun-relative speedup, for a 100-ensemble cluster, of 114.

For a wormhole routing network, when data items are exchanged between two ensembles, a round-trip message is required with estimated time, assuming a single hop is required, of perhaps 20 clock cycles. The estimated idealized Sun-relative speedup for the RRM would be very close to the cluster case. It is very possible that the network latency could be overlapped with other computation, and the increase in the total computation time compared with the cluster case should not be more than 20%.

5.3.4 Performance Estimates for Fluid Flow

Fluid dynamics can be studied using a 2D cellular automaton model [13]. This computational model is nearly ideal for the RRM, due to its very regular structure heavily using instructions that efficiently interchange bits among neighboring cells. The same communication pattern could be used for many other 2D processing and cellular automata problems. In fact, we have implemented Conway's game of Life

using these same techniques, and have achieved similar performance. Many other problems, such as certain vision algorithms, stress analysis and particle diffusion in solids, fit this pattern of computation.

We have implemented a version of the cellular automata approach based on a regular 2D hexagonal lattice. Each cell is connected to its six neighbors by links that may hold at most one particle traveling in each direction in each time step. We use unit time steps, unit particle masses, and unit velocity. Each particle is completely described by the link on which it currently resides, and all particles have constant kinetic energy and zero potential energy. At each time step, particles move along their links, possibly interact with other particles at the center of a hexagonal cell, and move to some other link.

We have implemented this model using one RRM cell to simulate each hexagonal cell of the model. Each RRM cell contains six bits that encode the presence or absence of outgoing particles on the links to its six neighbors. Communication is handled by transferring the six bits from each cell to the appropriate neighbor. Computation is handled by performing certain bitwise operations (such as `and`, `or`, `equal`) and a form of table lookup.

We used 1000 iterations of 529 hexagonal cells as the benchmark. Assuming that the ensemble chips will have a clock speed of 100 MHz, the whole benchmark should run in 2.2 to 2.6 ms. There are multiple ways to implement this problem in C for comparison. The fastest implementation we developed (using register declarations for variables, changing the way table lookup was handled, moving conditional expressions out of the main loop) ran in 1.4 seconds. This results in a Sun-relative speedup of between 400 and 670 for a single ensemble.

The instruction count for the main loop for this problem is about 220 instructions. We estimate that the communication overhead within a cluster, using neighbor-to-neighbor connections, could be as low as 48 clock cycles (6 bits $\times$ 4 cells per tile $\times$ 2). The transfers of marks between ensembles can take place in 12-bit parallel transfers (one cell for each tile on the edge of the ensemble). This gives 268 clock cycles per main loop or 2680 ns at 100 MHz. This gives a cluster-level performance that is 82% of ideal (= 220/268).

5.3.5 Performance Estimates for a Hardware Simulator

It is possible to do a simple kind of hardware simulation on the RRM extremely fast. The code to simulate two-input NAND and OR gates, where the output state of a gate is represented by the status of a specific mark, has only 24 instructions. This simple simulator cannot simulate arbitrary circuits, since there can be layout problems; gates must be close to the gates that produce their input signals. For a specific very simple circuit, comparison of the simulations with a highly optimized

C program running on a Sun workstation gives a Sun-relative speedup estimate for an ensemble of 533, and for an optimized C program that is a more general circuit simulator an estimated speedup of 1,500.

The cluster-level performance estimate is close to a linear scaleup of this. Only a single mark per edge cell needs to be transferred across the ensemble boundaries. It seems reasonable to assume that this could be done in 8 clock cycles. The cluster performance would then be 75% of ideal (= 24/32). The idealized Sun-relative speedup for a cluster would be 40,000 to 110,000.

5.3.6 Summary of Performance Estimates

For a 10,000-ensemble RRM, our present estimates are as follows:

- Raw peak performance: 576 trillion operations per second.
- For general symbolic applications (the numeric Fibonacci problem is taken as a typical example and the TAK function is a secondary example):
  - Ensemble Sun-relative speedup is roughly 6.7.
  - RRM performance with wormhole network at 88% efficiency gives an idealized Sun-relative speedup of 59,000.
- For highly regular symbolic applications (the sorting problem is taken as a typical example):
  - Ensemble performance is a Sun-relative speedup of 127.
  - Cluster-level performance is a Sun-relative speedup of 114.
  - RRM performance is estimated at over 80% efficiency (relative to the cluster performance) yielding a Sun-relative speedup of over 91.
- For systolic applications (a 2D fluid flow problem is taken as a typical example; a secondary example is a hardware simulator):
  - Ensemble performance is a Sun-relative speedup of 400 to 670.
  - Cluster-level performance, which should be attainable in practice, is 82% efficiency. This yields idealized Sun-relative speedups of 33,000 to 55,000.

References

[1] Hitoshi Aida, Joseph Goguen, and José Meseguer. Compiling concurrent rewriting onto the rewrite rule machine. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems, Montreal, Canada, June 1990*, pages 320–332. Springer LNCS 516, 1991.

[2] Hitoshi Aida, Sany Leinwand, and José Meseguer. Architectural design of the rewrite rule machine ensemble. In J. Delgado-Frias and W.R. Moore, editors, *VLSI for Artificial Intelligence and Neural Networks*, pages 11–22. Plenum Publ. Co., 1991. Proceedings of an International Workshop held in Oxford, England, September 1990.

[3] S. Borkar, R. Cohn, G. Cox, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWARP: an integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339. IEEE Press, 1988.

[4] Thinking Machines Corporation. Connection machine technical summary. 1990.

[5] William Dally. Network and processor architecture for message-driven computers. In R. Suaya and G. Birtwistle, editors, *VLSI and Parallel Computation*, pages 140–222. Morgan Kaufmann, 1990.

[6] J.A. Goguen. Semantic specifications for the rewrite rule machine. In A. Yonezawa, W. McColl, and T. Ito, editors, *Concurrency: Theory, Language and Architecture*, pages 216–234. Springer LNCS, Vol. 491, 1990.

[7] J.A. Goguen, S. Leinwand, J. Meseguer, and T. Winkler. The rewrite rule machine, 1988. Technical Report PRG-76, Oxford University, Programming Research Group, 1989.

[8] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy Winkler. An introduction to OBJ3. In Jean-Pierre Jouannaud and Stephane Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting, Orsay, France, July 8-10, 1987*, pages 258–263. Springer LNCS 308, 1988.

[9] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proc. Workshop on Graph Reduction, Santa Fe, New Mexico*, pages 53–93. Springer LNCS 279, 1987.

[10] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986.

[11] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[12] Joseph Goguen and José Meseguer. Software for the rewrite rule machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 628–637. ICOT, 1988.

[13] J. Hardy, B. Hasslacher, and Y. Pomeau. Lattice gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56:1505, 1986.

[14] W. Hillis. *The Connection Machine.* MIT Press, 1985.

[15] *HOT Chips III.* IEEE, 1991. Record of Symposium held at Stanford University August 26–27, 1991.

[16] S. Leinwand, J.A. Goguen, and T. Winkler. Cell and ensemble architecture for the rewrite rule machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 869–878. ICOT, 1988.

[17] José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA '90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.

[18] José Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur'90 Conference, Amsterdam, August 1990*, pages 384–400. Springer LNCS 458, 1990.

[19] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[20] José Meseguer and Timothy Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.

[21] Ulrich Schmidt and Knut Caesar. Datawave: A single-chip multiprocessor for video applications. *IEEE Micro*, 11(3):22–25, June 1991.

[22] Charles L. Seitz. Concurrent architectures. In R. Suaya and G. Birtwistle, editors, *VLSI and Parallel Computation*, pages 1–84. Morgan Kaufmann, 1990.

[23] Arthur Trew and Greg Wilson, editors. *Past, Present, Parallel: A Survey of Parallel Computing at the Beginning of the 1990s*. Springer-Verlag, 1991.

A Code of the Interensemble Simulator and of Several Benchmarks Run on it

The abstract interensemble simulator presented below was instrumental in our initial explorations of the network requirements for an RRM cluster. The abstract interensemble simulator essentially consists of an an annotated interpreter of rewrite-rules. This interpreter keeps track of which ensemble a term is supposed to reside in. Given a strategy for spawning tasks, the simulator then 'moves' terms from one ensemble to another (or causes newly allocated cells to be created in some other ensemble) by annotating the terms with the name (location) of a new ensemble.

Many experiments were run using this abstract simulator. For example, very preliminary experiments were conducted considering simple alternative cluster network configurations (2-D mesh, complete interconnect, and several bus-like connection layouts). Experiments were conducted with alternate strategies for spawning new tasks (allocating new cells in other ensembles only once one ensemble is nearly full, pushing out whole subterms when an ensemble becomes full, and static spawning routines). Network bandwidth requirements were estimated based on the number of pointers detected between terms residing on various simulated ensembles. Some initial experiments were also carried out regarding alternate formulations of the rewrite rules.

runtests

```
#!/bin/csh -f

#tst1 bubble sort
foreach size (5 10 15 20 25 30 35 40 50 60 70 80 90 100 110 120)
runtest tst1 $size
end

#tst2 peano fibo
foreach size (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
runtest tst2 $size
end

#tst3 merge sort
foreach size (5 10 15 20 25 30 35 40 50 60 70 80)
runtest tst3 $size
end

#tst4 numeric fibo
foreach size (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18)
runtest tst4 $size
end

#tst5 matrix multiply
foreach size (0 1 2 3 4 5 6 7 8 9)
runtest tst5 $size
end
```

runtest


```
#!/bin/csh -f
set file = $1
setenv SIZE $2
#alias ak /usr/dumas/tmp/bin/akcl
#alias ak /homes/dumas/obj/akcl/bin/akcl
alias ak /homes/roche/winkler/kcl4/akcl/unixport/saved_kcl
ak < $file.lsp
mv $file.out $file.$SIZE
```

make.lsp

```
(defvar setup-done nil)

(defun do-compile-file (file)
  (let* (#-SPARC #+KCL (compiler::*use-buggy* (member file '(
                "top/module_eval"
                "modexp/modexp_eval"
              ) :test #'equal))
         (nm (concatenate 'string file ".lsp"))
         (pn (probe-file nm)))
    (if pn
      (let* ((onm (concatenate 'string file ".o"))
             (opn (probe-file onm)))
        (if (or (not opn)
                (< (file-write-date onm) (file-write-date nm)))
            (progn
              (unless setup-done (do-setup) (setq setup-done t))
              (compile-file file)
              (terpri))
          (progn (princ "Already up to date: ") (princ file) (terpri)))
        'done)
      (progn (princ "No such lisp file as: ") (princ file) (terpri)
             'err)))
  )

(setq file-list '(
"a"
"alloc4"
"ies"
"patch"
"strat"
"strat2"
"filecol"
"a2"
))

(defun do-setup () (dolist (f file-list) (load f)))

(defun cmp ()
  (dolist (f file-list) (do-compile-file f))
  )

(cmp)
```

```
(defun quo (x y) (coerce (/ x y) 'float))

(defun ensiostats ()
  (shm)
  (princ "global ensemble I/O state") (terpri)
  (let ((gsi (stats0_create))
        (gso (stats0_create))
        ;(none (stats_make))
        )
  (dotimes (i rrm_size)
    (let ((ss (stats_ensemble i)))
    (unless (not (stats_used ss)) ;(equal none ss)
      (merge_stats (stats0_glbl (stats_input ss)) gsi)
      (merge_stats (stats0_glbl (stats_output ss)) gso)
    )))
  (stats0_compute_derived_basic gsi)
  (stats0_compute_derived_basic gso)
  (princ "input") (terpri)
  (stats0_print gsi) (terpri)
  (princ "output") (terpri)
  (stats0_print gso) (terpri)
  (let ((globi (stats0_glbl input_stats))
        (globo (stats0_glbl output_stats)))
  (shm)
  (princ "global per step I/O state") (terpri)
  (stats0_compute_derived_basic globi)
  (stats0_compute_derived_basic globo)
  (princ "input") (terpri)
  (stats0_print globi) (terpri)
  (princ "output") (terpri)
  (stats0_print globo) (terpri))
  ))

(defun stats0_print (x)
  (princ "num: ") (prin1 (stats0_num x)) (terpri)
  (princ "sum: ") (prin1 (stats0_sum x)) (terpri)
  (princ "min: ") (prin1 (stats0_min x)) (terpri)
  (princ "max: ") (prin1 (stats0_max x)) (terpri)
  ; (princ "sum_sq: ") (prin1 (stats0_sum_sq x)) (terpri)
  (princ "av: ") (format t "~6,2f" (stats0_av x)) (terpri)
  (princ "dev: ") (format t "~6,2f" (stats0_dev x)) (terpri)
  (let ((zer (or (cdr (assoc 0 (stats0_distr x))) 0)))
    (when (and (< 0 zer) (< zer (stats0_num x)))
      (princ "non zero av: ")
      (format t "~6,2f" (quo (stats0_sum x) (- (stats0_num x) zer))) (terpri)
    ))
  (princ "disribution: ") (terpri)
    (print_distr (stats0_distr x))
  )

(defun stats0_compute_derived_basic (ss)
  (let ((n (stats0_num ss))
        (sum (stats0_sum ss))
        (sum_sq (stats0_sum_sq ss)))
  (if (= 0 n)
    (progn
      (setf (stats0_av ss) 0.0)
      (setf (stats0_dev ss) 0.0))
  (let ((av (coerce-float (/ sum n))))
  (setf (stats0_av ss) av)
  (setf (stats0_dev ss)
      (sqrt (- (coerce-float (/ sum_sq n)) (* av av))))

  ))))

(defun print_distr (d)
  (let ((sum 0) (zers 0))
  (dolist (e d)
    (when (zerop (car e)) (setq zers (cdr e)))
    (incf sum (cdr e)))
  (let ((nzsum (- sum zers)) (nzcum 0.0) (cum 0.0))
  (dolist (e d)
    (let ((nzval (if (zerop (car e)) 0.0 (quo (cdr e) nzsum)))
          (val (quo (cdr e) sum)))
    (setq nzcum (+ nzcum nzval))
    (setq cum (+ cum val))
    (format t "~5d: ~6d ~8,2f ~8,2f ~8,2f~%"
       (car e) (cdr e) (* 100 val) (* 100 nzcum) (* 100 cum))
))))))
```

```
(defvar orig_ens)

(defun term_top_size (x)
  (if (or (term_is_var x) (term_is_const x)) 1
    (let ((res 1)) ;count one for the top node
      (dolist (e (term_args x))
        (when (occurs_in_ensemble e orig_ens)
          (setq res (+ res (term_top_size e))))
      )
      res
    )
  ))
  (and (not (is_root e)) ...)

(defun compute_ens_size (e)
  (let ((orig_ens e))
  (let ((res 0)
        (inf (status_ensemble e)))
    (dolist (r (status_roots inf))
      (setq res (+ res (term_top_size r)))
    )
    res
  )))
```

```
; Matrix Multiply
(unless (fboundp 'simul) (load "iss"))

(to-file "tst5.out"

(setq tst 5)
(setq test-size 3)
(setq verbose 0)
(setq intra_cluster_penalty 2)
(setq inter_cluster_penalty 2)
(setq push_lim 3)
(setq build_lim 3)
(load "p")

(init)

(setq all_partitions '(
(
(eq (rowxtr nil a (empty-matrix nil)) (empty-row nil))
(eq (rowxtr nil a (alm nil b n))
    (al nil (ip nil a b) (rowxtr nil a n)))

(eq (mmtr nil (empty-matrix nil) n) (empty-matrix nil))
(eq (mmtr nil (alm nil a m) n)
    (alm nil (rowxtr nil a n) (mmtr nil m n)))

(eq (*m nil m n) (mmtr nil m (tr nil n)))
;(eq (sq nil m) (*m nil m m))
)
(
(ceq (* nil e1 e2) (b-i (mak (* (car (val 'e1)) (car (val 'e2)))))
     (and nil (numberp nil e1) (numberp nil e2)))
(ceq (+ nil e1 e2) (b-i (mak (+ (car (val 'e1)) (car (val 'e2)))))
     (and nil (numberp nil e1) (numberp nil e2)))

(eq (ip nil (empty-row nil) a) (0 nil))
(eq (ip nil a (empty-row nil)) (0 nil))
(eq (ip nil (al nil x a) (al nil y b))
    (+ nil (* nil x y) (ip nil a b)))

(eq (al nil (0 nil) (empty-row nil)) (empty-row nil))
(eq (alm nil (empty-row nil) (empty-matrix nil)) (empty-matrix nil))
)
(
(eq (firsts nil (empty-matrix nil)) (empty-row nil))
(eq (firsts nil (alm nil (empty-row nil) m))
    (al nil (0 nil) (firsts nil m)))
(eq (firsts nil (alm nil (al nil i r) m)) (al nil i (firsts nil m)))

(eq (rests nil (empty-matrix nil)) (empty-matrix nil))
(eq (rests nil (alm nil (empty-row nil) m))
    (alm nil (empty-row nil) (rests nil m)))
(eq (rests nil (alm nil (al nil i r) m)) (alm nil r (rests nil m)))

(eq (tr nil (empty-matrix nil)) (empty-matrix nil))
;(ceq (tr nil m) (alm nil (firsts nil m) (tr nil (rests nil m)))
;     (not nil (== nil m '(empty-matrix nil))))
(eq (tr nil (alm nil (al nil i r) m))
    (alm nil (firsts nil (alm nil (al nil i r) m))
             (tr nil (rests nil (alm nil (al nil i r) m)))))
)
))

(let ((clock 0) (mat '(empty-matrix)))
(dotimes (ic test-size)
  (let ((i (- test-size ic 1))
        (row '(empty-row)))
  (dotimes (jc test-size)
    (let ((j (- test-size jc 1)))
      (setq row (list 'al (list (+ i j)) row))
  ))
  (setq mat (list 'alm row mat))
))
(setq root
    (init_term (rrm_ensemble 0 0 1 1)
               (list '*m mat mat)))
)

(princ "------------------------------------------------------------")
(princ " initial") (terpri)
(print_term root) (terpri)

(simul)

(load "a")
(v)
)
```

```
;;; Copyright 1988 SRI International
;; DANGER -- this is implementation dependent
;; This used to get nice output line lengths; see replacement below

#+KCL
(defCfun "object filecol(x) object x;" 0
" Creturn(make_fixnum(file_column(x)));"
)

#+KCL
(defentry filecol (object) (object filecol))

#-LUCID
#+GENERIC
(defun filecol (x) 0) ; use this if you cannot define as

#+LUCID
(defun filecol (x) (lucid::calculate-output-column x))

(defun file-column (strm)
  (if (eq 'stream (type-of strm))
    (filecol strm)
    0))
```

```
;------------------------------------------------------------------------
(defvar size_lim 100)
(defvar reloc_size_lim size_lim)
(defvar ens_full_thres size_lim)
(defvar conflict_reloc_size 0)
(defvar ens_size_low 50)
(defvar ens_full_push 0)
(defvar ens_full_push_single 0)

(defvar mssg_data_size 100)
(defvar mssg_reloc_req_size 40)
(defvar mssg_push_size 120)

(defvar step_input 0)
(defvar step_output 0)

(defvar input_state)
(defvar output_state)

;------------------------------------------------------------------------
(defvar io_actions nil)

(defun io_action (dir e val)
  (if (eq 'in dir)
    (progn (ens_add_input e val) (incf step_input val))
  (if (eq 'out dir)
    (progn (ens_add_output e val) (incf step_output val))
    (break "io_action error")
  )))

(defun perform_io_action (x)
  (io_action (car x) (cadr x) (caddr x)))

(defun perform_io_actions ()
  (let ((new nil))
  (dolist (a io_actions)
    (if (<= (car a) clock)
      (perform_io_action (cdr a))
      (push a new))
    )
  (setq io_actions new)
  ))

(defun add_io_action (time dir e val)
  (push (list time dir e val) io_actions))

;------------------------------------------------------------------------
(defun global_io_new_group ()
  ; (format t " ~8d ~8d~%" step_output step_input)
  (insert_val input_state step_input)
  (new_group input_state)
  (setq step_input 0)

  (insert_val output_state step_output)
  (new_group output_state)
  (setq step_output 0)

  (perform_io_actions)
  )

;------------------------------------------------------------------------
(defun compute_reloc_penalty (to from knd)
  (if (eq 'near knd)
    intra_cluster_penalty
    inter_cluster_penalty))

(defun compute_push_penalty (to from)
  (if (= (aref ensemble_cluster to) (aref ensemble_cluster from))
    intra_cluster_penalty
    inter_cluster_penalty))

(defun force_relocation (x e)
  (setq forced_relocation t)
  (setq some_relocation t)
  (let ((loc (term_loc x)))
  (when loc ;@@ wait if being relocated
  (let ((kind (if (= (aref ensemble_cluster loc) (aref ensemble_cluster e))
                  'near 'far)))
  (let ((delay (compute_reloc_penalty e loc kind)))
  (let ((wh (+ clock delay)))
  (let ((res (assoc e (get_an x 'where))))
    (unless (and res (<= (cdr res) wh))
      (when (and verbose (<= 4 verbose))
        (princ "### ") (prin1 kind) (princ " reloc") (terpri)
        (princ "to ") (prin1 e)
        (princ " at ") (prin1 wh) (terpri)
        (princ "data ") (print_term_choose x) (terpri)
      )
      (if (or (not test_comm) (test_ensemble_reloc e loc))
        (if (< reloc_size_lim (ens_size e))
          (progn
            (when (<= 5 verbose)
              (princ "### reloc size conflict ens: ")
              (prin1 e) (princ " from ") (prin1 loc) (terpri))
            (incf conflict_reloc_size))
          (progn
            (term_add_loc x e wh)
            (if (eq 'near kind)
              (progn
                (ens_inc_reloc e)
                (io_action 'out e mssg_reloc_req_size)
                (let ((mid (+ clock (truncate delay 2))))
                (add_io_action mid 'in loc mssg_reloc_req_size)
                (add_io_action mid 'out loc mssg_data_size))
                (add_io_action wh 'in e mssg_data_size)
                (cluster_ensemble_inc_reloc e loc))
              (progn
                (ens_inc_reloc_far e)
                (io_action 'out e mssg_reloc_req_size)
                (let ((mid (+ clock (truncate delay 2))))
                (add_io_action mid 'in loc mssg_reloc_req_size)
                (add_io_action mid 'out loc mssg_data_size))
                (add_io_action wh 'in e mssg_data_size)
                (cluster_inc_reloc
                 (aref ensemble_cluster e)
                 (aref ensemble_cluster loc))))
            (when (and res (< wh (cdr res))) (setf (cdr res) wh)))) ;@ note
        (progn
          (when (<= 5 verbose)
            (princ "### reloc comm conflict ens: ")
            (prin1 e) (princ " and ") (prin1 loc) (terpri))
          (if (eq 'near kind)
            (incf conflict_near)
            (incf conflict_far))) ;@ note
      ) ; if or not test_comm
    )
  )))))))

;------------------------------------------------------------------------
; if ensemble e is full or i is big enough, push to another ensemble
; per-rule treatment
(defun perform_replacement (e x r i y)
  (when (and verbose (<= 2 verbose))
    (when (<= 5 verbose)
      (princ "#######################################################") (terpri))
    (princ "### repl")
    (princ "  ens ") (prin1 e)
    (princ " level ") (prin1 i) (terpri)
    (princ "term ") (print_term_choose x) (terpri)
    (princ "repl ") (print_term_choose y) (terpri)
    (princ "rule ") (prin1 r) (terpri)
  )
  (let ((current_ensemble e))
  (replace_list x (term_remake y))
      ; term_remake -- need this in case rule is of form l = x
      ;  result is always in the original ensemble
  (if (or (<= push_lim i)
          (< size_lim (ens_size_recompute e)))
    (progn
      (move_node x e)
```

```
      (dolist (u (term_args x))
        (when (is_new u)
          (push_node u (alloc_ensemble e))
          (dolist (m (term_args u))
            (update_new_loc m e))
      ))
      (ens_size_invalidate e)
      )
    (update_new_loc x e))
  ))

(defun ens_size_invalidate (e)
  (let ((inf (status_ensemble e)))
  (let ((x (status_size inf)))
    (when x
      (setf (car x) -1)))))

(defun push_node (u e)
  (let ((res (assoc 'new_where (term_an u))))
  (when res
    (when (and verbose (<= 2 verbose))
      (princ "### pushing ")
      (prinl current_ensemble)
      (princ " to ")
      (prinl e) (terpri)
      (princ "pushed ") (print_term_choose u) (terpri))
    (setf (car res) 'where)
    (let ((wh (+ clock (compute_push_penalty e current_ensemble))))
    (setf (cadr res) (cons e wh)) ;@@ note
    (ens_inc_push current_ensemble)
    (io_action 'out current_ensemble msg_push_size)
    (add_io_action wh 'in e msg_push_size)
    (cluster_ensemble_inc_push current_ensemble e) ;@@new
    (ensemble_add_root e u)
    (ens_size_invalidate e) ;@@
  )
  )))

; flg: should this be a root?
(defun build_term (flg e i x)
  (if (or (term_is_var x) (term_is_bi x)) x
    (let ((res nil)
          (i2 (+ i 1)) (push nil))
    (when (<= build_lim i) (setq i2 0  push t))
    (dolist (a (cdr x))
      (push (build_term push (if push (alloc_ensemble e) e) i2 a) res)
    )
    (let ((res (term_make_where e (car x) (nreverse res))))
    (when flg
      (ensemble_add_root e res)
      (ens_size_invalidate e)) ;@@
    res
  ))))

;-----------------------------------------------------------------------
(defvar test_comm nil)
(defvar intra_cluster_lim 10)
(defvar inter_cluster_lim 10)
(defvar conflict_near 0)
(defvar conflict_far 0)

(defun test_ensemble_reloc (i j)
  (let ((ic (aref ensemble_cluster i)) (jc (aref ensemble_cluster j)))
  (if (= ic jc)
    (let ((cesi (aref cluster_ensemble_stats ic)))
      (or
       (null cesi)
       (< (stats0_cur (aref cesi (cluster_ensemble_number i)
                                 (cluster_ensemble_number j)))
          intra_cluster_lim)))
    (< (stats0_cur (aref cluster_stats ic jc)) inter_cluster_lim))
  ))
;-----------------------------------------------------------------------
(defvar orig_ens)

(defun term_top_size (x)
  (if (or (term_is_var x) (term_is_const x)) 1
    (let ((res 1)) ;count one for the top node
      (dolist (e (term_args x))
        (when (occure_in_ensemble e orig_ens)
          (setq res (+ res (term_top_size e)))))
      )
      res
    )
  ))
; (and (not (is_root e)) ...)

(defun compute_ens_size (e)
  (let ((orig_ens e))
  (let ((res 0)
        (inf (status_ensemble e)))
    (dolist (r (status_roots inf))
      (setq res (+ res (term_top_size r)))
    )
    res
  )))
;-----------------------------------------------------------------------
(defun ens_full_strategy (e)
  (let ((inf (status_ensemble e))
        (sz (ens_size e))
        (orig_ens e))
  (when (< ens_full_thres sz)
    (incf ens_full_push)
    (let ((lst (status_roots inf)))
    (if (null (cdr lst))
      ; push part of the one top term
      (let ((trm (car lst)))
      (let ((args (term_args trm)))
        (when (<= 5 verbose)
          (princ "### ens full pushing part ") (prinl e)
          (princ "  size: ") (prinl sz) (terpri))
        (incf ens_full_push_single)
        (loop
         (when (cdr args) (return))
         (setq args (term_args (car args)))
        )
        (loop
         (setq sz (- sz (term_top_size (car args))))
         (when (<= 5 verbose)
           (princ "###### ens full pushing part ")
           (print_term_choose (car args))
           (terpri))
         (shove_node e (car args))
         (when (or (null (cdr args)) (< sz ens_size_low)) (return))
         (setq args (cdr args))
        )
      ))
    (progn
      (loop
       (setq sz (- sz (term_top_size (car lst))))
       (shove_node e (car lst))
       (when (<= 5 verbose)
         (princ "### ens full pushing ")
         (prinl e) (princ " ") (print_term_choose (car lst)) (terpri))
       (when (or (null (cdr lst)) (< sz ens_size_low)) (return)) ;moved down
       (setq lst (cdr lst))
      )
      (set_status_roots inf lst)
    ))
  ))
      .
(defun shove_node (e trm)
  (let ((newens (alloc_ensemble e)))
  (term_add_loc trm newens (compute_push_penalty newens e))
  (term_remove_loc trm e)
  (ensemble_add_root newens trm)
  (ens_size_invalidate newens)
  (ens_inc_push e) ;@
  (cluster_ensemble_inc_push e newens) ;@
```

```
  ))

(defun term_remove_loc (x e)
  (let ((wh (get_an x 'where)))
  (let ((res nil))
  (dolist (pr wh)
    (unless (= e (car pr)) (push pr res)))
  (set_an x 'where (nreverse res))
  )))

(defun term_remove_loc_rooted (x e)
  (unless (is_root x)
    (unless (or (term_is_var x) (numberp x) (term_is_bi x))
      (term_remove_loc x e)
      (dolist (xe (term_args x))
        (term_remove_loc_rooted xe e))
      )
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar print_term_choice nil)

(defun print_term_choose (x)
  (if print_term_choice (print_term x)
    (print_term_partial 3 x)))

(defun print_term_partial (d x)
  (when (< 60 (col)) (terpri) (princ "  "))
  (if (< d 1) (princ "#")
  (if (term_is_var x) (prin1 x)
  (if (or (numberp x) (term_is_bi x)) (prin1 (term_const_val x))
  (progn
    (princ "(")
    (prin1 (term_op x))
    (let ((dm1 (- d 1)))
    (dolist (e (term_args x))
      (princ " ")
      (when (is_root e) (princ "[") (prin1 (term_root_loc e)) (princ "]"))
      (print_term_partial dm1 e)))
    (when (< 60 (col)) (terpri) (princ "  "))
    (princ ")")
  )))))

(defun term_root_loc (x)
  (let ((locs (get_an x 'where)))
    (if locs
        (dolist (e locs)
          (when (and (<= (cdr e) clock)
                     (member x (status_roots (status_ensemble (car e)))))
            (return (car e)))
        )
      -1)
  ))
```

```
(princ "rrm size: rrm clusters ")
  (prin1 cluster_height) (princ "x") (prin1 cluster_width)
  (princ "  cluster ensembles ")
  (prin1 rrm_cluster_height) (princ "x") (prin1 rrm_cluster_width) (terpri)

(princ "stats: ens ") (prin1 ens_stats_param)
  (princ "  clust ens ") (prin1 clust_ens_stats_param)
  (princ "  clust ") (prin1 clust_param) (terpri)

(setq intra_cluster_penalty 2)
(princ "intra cluster penalty: ") (princ intra_cluster_penalty) (terpri)

(setq inter_cluster_penalty 2) ;@20
(princ "inter cluster penalty: ") (princ inter_cluster_penalty) (terpri)

(setq gc_period 1)
(princ "gc period: ") (princ gc_period) (terpri)

(unless (boundp 'alloc_range) (load "alloc4"))
(princ "alloc threshold: ") (prin1 alloc_thres) (terpri)
(princ "alloc randomization range: ") (prin1 alloc_range)
  (princ "    cut: ") (prin1 alloc_cut) (terpri)
(princ "when totally full: alloc ")
  (princ (if alloc_totally_full_random "at random" "in local cluster"))
  (terpri)

(unless (boundp 'compute_push_penalty)
  (load "strat"))

(when (boundp 'test_comm)
(setq test_comm t)
(if test_comm
(progn
(setq inter_cluster_lim 40) ;@100
(princ "intra cluster comm limit: ") (prin1 intra_cluster_lim) (terpri)
(princ "inter cluster comm limit: ") (prin1 inter_cluster_lim) (terpri))
(progn
(princ "no communication limits") (terpri))
))

(setq size_lim 100)
(princ "size limit: ") (prin1 size_lim) (terpri)
(when (boundp 'reloc_size_lim)
(setq reloc_size_lim 100)
(princ "reloc size limit: ") (prin1 reloc_size_lim) (terpri))

(when (boundp 'ens_full_thres)
(setq ens_full_thres 100)
(princ "ensemble full threshold: ") (prin1 ens_full_thres) (terpri)
(setq ens_size_low 55)
(princ "ensemble low level: ") (prin1 ens_size_low) (terpri))

(defvar lim 20)
(setq push_lim lim)
(princ "push lim: ") (prin1 push_lim)

(setq build_lim lim)
(princ "    build lim: ") (prin1 build_lim) (terpri)

(let ((val (si:getenv "SIZE")))
(when val
  (setq test-size (read-from-string val))
  )
)
(princ "problem size: ") (prin1 test-size) (terpri)

(when (boundp 'clock_limit)
  (princ "clock limit: ") (prin1 clock_limit) (terpri))

(when nil
(setq *random-state* (make-random-state t))
(princ "random state: ") (prin1 *random-state*) (terpri))

(load "patch")
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Inter ensemble Simulations
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;SL: GC is equivalent to using just the term, then a pass to find current
;   roots of ensembles
;@@todo reduced testing (delay)
;   simple approach: per ensemble flag, clear whenever start set of
;   all partitions, set when any rule is applied.  If is not set in a pass
;   all nodes with reduced children become reduced (??)
;@@todo random walk doesn't give enough dispersion
;@@ GC where's too, if no longer there, then remove from where list
;minor randomization in penalties?
;change names?? -- will try adding alternate versions
;   (ens_inc_reloc e)  --> (ens_inc_comm e)
;   (cluster_ensemble_inc_reloc e loc) --> (cluster_ensemble_inc_comm x y)
;   (cluster_inc_reloc x y) --> (cluster_inc_comm x y)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; terms labelled with partitions that they occur in and time
;   when valid
; @@problem: really would like to have each pointer be to a term at a location

; each ensemble has
;   a current rule set and current rule
;   a current set of term roots
;   a current "full-ness"
; use of term in another ensemble causes current-time plus delay
;   to be added as time for desired ensemble

; each ensemble is in a cluster
; communication is inside a cluster or across clusters
; stats on all communication across clusters are kept
; communication across clusters takes even longer

; basic simulation step: activate all ensembles
;   -- apply current rule of rules set and find next rule
; if have to switch rule sets, then increment an ensemble counter

; annotations on term -- ensembles having copies, reduced status
; for reduced: keep track of set of rules used on term since last success
; tags: where reduced rules

; probably want to carefully do match and replacement:
; replacement is delayed, put on global action list, done at end
; of major simulation step

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar verbose 1)
(defvar periodic 500)
(defvar show_term nil)
(setq *print-case* :downcase)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar cluster_height 4)
(defvar cluster_width 4)
(defvar cluster_size (* cluster_height cluster_width))
(defvar rrm_cluster_height 4)
(defvar rrm_cluster_width 4)
(defvar rrm_height (* cluster_height rrm_cluster_height))
(defvar rrm_width (* cluster_width rrm_cluster_width))
(defvar rrm_cluster_size (* rrm_cluster_height rrm_cluster_width))
(defvar rrm_size (* rrm_cluster_height rrm_cluster_width cluster_size))
(defvar intra_cluster_penalty 5)
(defvar inter_cluster_penalty 10)
(defvar all_partitions nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar push_lim 5)
(defvar build_lim 5)
(defvar gc_period 10)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar root)
(defvar current_ensemble)
(defvar ensemble_cluster)
(defvar ensemble_status)
(defvar cluster_stats)
(defvar cluster_ensemble_stats)
(defvar ensemble_stats)
(defvar cluster_ensemble_stats2)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar clock 0)
(defvar action_list nil)

(defvar clock_limit 999999)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar rule_successful nil)
(defvar forced_relocation nil)
(defvar some_relocation nil)
(defvar replacements 0)
(defvar replacement_stats)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar ens_stats_param 10)
(defvar clust_ens_stats_param 10)
(defvar clust_param 10)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun cluster_row (e)
  (rem (truncate e rrm_width) cluster_height))
(defun cluster_col (e)
  (rem e cluster_width))

(defun cluster_ensemble_number (e)
  (+ (* (cluster_row e) cluster_width)
     (cluster_col e)))

;(defun cluster_elt (c i j)
;  (rrm_ensemble
;   (truncate c rrm_cluster_width) (rem c rrm_cluster_width)
;   i j)
;  )
(defun cluster_elt (c i j)
  (let ((rm (rem c rrm_cluster_width)))
  (+ (* (+  (* (- c rm) cluster_height) rm) cluster_width)
     (* i rrm_width)
     j))
  )

(defun ensemble_number (i j)
  (+ (* i cluster_width) j))

(defun rrm_row (e)
  (truncate e rrm_width))
(defun rrm_col (e)
  (rem e rrm_width))

(defun rrm_cluster (e)
  (+ (truncate (rem e rrm_width) cluster_width)
     (* (truncate (truncate e rrm_width) cluster_height)
        rrm_cluster_width)))

; cluster row and col (within rrm); ensemble row and col (within cluster)
(defun rrm_ensemble (i j k l)
  (+ (* i rrm_width cluster_height)
     (* j cluster_width)
     (* k rrm_width)
     l))

(defun rrm_elt (i j)
  (+ (* i rrm_width)
     j))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; intra-cluster
(defun cluster_left (e)
  (let ((col (rem e cluster_width)))
    (+ e (- col) (if (= col 0) (- cluster_width 1) (- col 1)))))
```

```
(defun cluster_right (e)
  (let ((col (rem e cluster_width)))
    (+ e (- col) (if (= col (- cluster_width 1)) 0 (+ col 1)))))

(defun cluster_up (e)
  (let ((row (rem (truncate e rrm_width) cluster_height)))
    (+ e (* rrm_width (if (= row 0) (- cluster_height row 1) -1)))))

(defun cluster_down (e)
  (let ((row (rem (truncate e rrm_width) cluster_height)))
    (+ e (* rrm_width
            (if (= row (- cluster_height 1)) (- row) 1)))))

;-----------------------------------------------------------------------
; inter-cluster
(defun rrm_left (e)
  (let ((col (rem e rrm_width)))
    (+ e (- col) (if (= col 0) (- rrm_width 1) (- col 1)))))

(defun rrm_right (e)
  (let ((col (rem e rrm_width)))
    (+ e (- col)
       (if (= col (- rrm_width 1)) 0 (+ col 1)))))

(defun rrm_up (e)
  (rem (+ e rrm_size (- rrm_width)) rrm_size))

(defun rrm_down (e)
  (rem (+ e rrm_width) rrm_size))

;-----------------------------------------------------------------------
(defun reset_clock ()
  (setq clock 0))

(defun tick ()
  (all_new_group)
  (setq clock (+ clock 1)))

;-----------------------------------------------------------------------
(defun add_repl (x r i y)
  (push (list 'repl current_ensemble x r i y) action_list))

(defun replace_list (x y)
  (setf (car x) (car y))
  (setf (cdr x) (cdr y))
  x)

(defun add_root (e x)
  (push (list 'root e x) action_list))

(defun ensemble_add_root (e x)
  (ens_set_used e)
  (let ((inf (status_ensemble e)))
    (set_status_roots inf
      (cons x (status_roots inf)))))

(defun perform_actions ()
  (dolist (a action_list)
    (if (eq 'repl (car a))
      (apply #'perform_replacement (cdr a))
    (if (eq 'root (car a))
        (ensemble_add_root (cadr a) (caddr a))
      (break "perform_actions: unknown action"))
  )))

;-----------------------------------------------------------------------
;(defun term_make_nil (x y) (cons x (cons nil y))) ; not used
(defun term_make_an (x a y) (cons x (cons a y)))
(defun term_args (x) (cddr x))
(defun term_op (x) (car x))
(defun term_an (x) (cadr x))
(defun term_is_var (x) (and x (symbolp x)))
(defun term_is_const (x)
  (or (numberp x)
      (and (consp x)
           (or (numberp (car x))
               (eq 'bi (car x))))))
(defun term_is_bi (x)
  (and (consp x) (eq 'bi (car x))))
(defun term_const_val (x)
  (if (numberp x) x
  (if (and (consp x) (numberp (car x))) (car x)
  (if (and (consp x) (eq 'bi (car x)))
      (cadr x)
    (break "term_const_val")))))

(defun get_an (x n)
  (cdr (assoc n (term_an x)))
  )
(defun set_an (x n v)
  (let ((res (assoc n (term_an x))))
    (if res (setf (cdr res) v)
      (setf (cadr x) (cons (cons n v) (term_an x)))) ;@@
  ))
(defun an_add (a n v)
  (cons (cons n v) a))

;@@ note handling of where
;@@not used
(defun term_make (x y)
  (cons x (cons
            (list
             (cons 'new_where (list (cons current_ensemble clock)))
             )
            y)))

(defun term_make_direct (x y)
  (cons x (cons
            (list
             (cons 'where (list (cons current_ensemble clock)))
             )
            y)))

(defun term_make_where (wh x y)
  (cons x (cons
            (list
             (cons 'where (list (cons wh clock)))
             )
            y)))

;@@keep more annotations? -- don't want where
(defun term_remake (x)
  (cons (car x)
        (cons
         (list
          (cons 'new_where (list (cons current_ensemble clock)))
          )
         (cddr x))))

;-----------------------------------------------------------------------
(defun term_expand (x)
  (if (term_is_var x) x
    (let ((res nil))
    (dolist (a (cdr x))
      (push (term_expand a) res)
    )
    (term_make_direct (car x) (nreverse res))
  )))

;-----------------------------------------------------------------------
(defun occurs_in_ensemble (x e)
  (let ((locs (get_an x 'where)))
  (let ((res (assoc e locs)))
    (and res
         (<= (cdr res) clock))
  )))

;@@ really would like to have each pointer be to a term at a location
(defun term_loc (x)
  (let ((locs (get_an x 'where)))
```

```
    (if locs
        (dolist (e locs)
          (when (<= (cdr e) clock) (return (car e)))
        )
      nil)
  ))

(defun term_add_loc (x e w)
  (let ((wh (get_an x 'where)))
  (let ((res (assoc e wh)))
    (if res
        (when (< w (cdr res))
          (setf (cdr res) w))
      (set_an x 'where (cons (cons e w) wh)))
  )))

(defun an_add_loc (a e w)
  (an_add a 'where (list (cons e w))))

;-----------------------------------------------------------------------
; uses: current_ensemble
(defun term_match (p x)
  (when (and verbose (<= 10 verbose))
    (princ "### match ")
    (print_term_choose p) (princ " --- ")
    (print_term_choose x) (terpri))
  (let ((res (term_match_1 p x)))
  (when (and verbose (<= 10 verbose))
    (princ "### match      result ")
    (prinl res) (terpri))
  res
  ))

(defun term_match_1 (p x)
 (if (term_is_var p)
      (if (term_is_var x) (list (list p x))
      (if (not (occurs_in_ensemble x current_ensemble))
          (progn (force_relocation x current_ensemble) 'fail)
        (list (list p x))))
  (if (term_is_var x) 'fail
  (if (term_is_const p)
      (if (atom p) (if (equal p x) nil 'fail)
        (if (and (consp x) (equal (car p) (car x))) nil 'fail))
  (if (term_is_const x) 'fail
  (if (not (occurs_in_ensemble x current_ensemble))
      (progn (force_relocation x current_ensemble) 'fail)
  (if (not (equal (term_op p) (term_op x))) 'fail
    (let ((res nil) (pl (term_args p)) (xl (term_args x)) (val nil))
      (loop
       (when (null pl) (if (null xl) (return res) (return 'fail)))
       (when (null xl) (return 'fail))
       (setq val (term_match_1 (car pl) (car xl)))
       (if (eq 'fail val) (return 'fail)
         (setq res (append res val)))
       (setq pl (cdr pl)  xl (cdr xl))
      )
  ))))))))

;-----------------------------------------------------------------------
(defun print_term (x)
  (when (< 60 (col)) (terpri) (princ "  "))
  (if (term_is_var x) (prinl x)
  (if (term_is_const x) (prinl (term_const_val x))
  (progn
    (princ "(")
    (prinl (term_op x))
    (dolist (e (term_args x))
      (princ " ")
      (print_term e))
    (when (< 60 (col)) (terpri) (princ "  "))
    (princ ")")
  ))))

;-----------------------------------------------------------------------
(defvar subs)

(defun term_inst (x s)
  (if (term_is_var x)
      (let ((res (assoc x s)))
        (if res (cadr res)
          x))
  (if (or (atom x) (term_is_bi x)) x
  (if (and (consp x) (eq 'b-i (car x)))
      (let ((subs s))
        (eval (cadr x)))
  (term_make_an (term_op x) ;@@keeping annotations
;             (an_add_loc (term_an x) current_ensemble clock)
              (an_add (term_an x)
                      'new_where
                      (list (cons current_ensemble clock)))
              (mapcar #'(lambda (u) (term_inst u s)) (term_args x)))
  ))))

(defun var_inst (v s)
  (let ((res (assoc v s)))
    (if res (cadr res)
      nil))
  )

(defun val (v) (var_inst v subs))

(defun mak (x) (list x))

;-----------------------------------------------------------------------
(defun rule_lhs (x) (cadr x))
(defun rule_rhs (x) (caddr x))
(defun rule_cond (x) (cadddr x))

;-----------------------------------------------------------------------
(defun rule_inst (r x mt)
  (if (eq 'b-i (rule_rhs r))
      (bi_inst r x mt)
    (term_inst (rule_rhs r) mt))
  )

(defun bi_inst (r x mt)
  (apply (car x) (cddr x)) ;@@??
  )

(defun eval_cond_direct (e mt)
  (if (term_is_var e)
      (let ((res (assoc e mt)))
        (if res (direct_value (cadr res))
          (break "eval_cond_direct: unbound var in condition")))
  (if (and (consp e) (eq 'quote (car e))) (cadr e)
  (if (term_is_const e) (term_const_val e)
  (if (member (term_op e) '(= < <= > >= not))
      (let ((as nil))
        (dolist (x (term_args e)
                   (apply (term_op e) (nreverse as)))
          (let ((val (eval_cond_direct x mt)))
          (when (eq 'fail val) (return 'fail))
          (push val as))
        )
      )
  (if (eq 'and (term_op e))
      (let ((al (eval_cond_direct (car (term_args e)) mt)))
        (if (eq 'fail al) 'fail
        (if (null al) nil
          (eval_cond_direct (cadr (term_args e)) mt))))
  (if (eq 'or (term_op e))
      (let ((al (eval_cond_direct (car (term_args e)) mt)))
        (if (eq 'fail al) 'fail
        (if (eq t al) t
          (eval_cond_direct (cadr (term_args e)) mt))))
  (if (eq '== (term_op e))
      (term_equal (eval_cond_direct (car (term_args e)) mt)
                  (eval_cond_direct (cadr (term_args e)) mt))
  (if (member (term_op e) '(numberp equal atom nequal))
      (let ((as nil))
```

```
        (dolist (x (term_args e))
          (push (eval_cond_direct x mt) as))
        (apply (term_op e) (nreverse as)))
      'fail
  )))))))))

(defun direct_value (x)
  (if (term_is_const x) (term_const_val x)
    x))

(defun nequal (x y) (not (equal x y)))

(defun term_equal (x y)
  (if (atom x) (equal x y)
  (if (term_is_bi x) (equal x y)
  (if (term_is_bi y) nil
  (and (equal (term_op x) (term_op y))
       (let ((xa (term_args x)) (ya (term_args y)))
       (loop
         (when (null xa) (return (null ya)))
         (unless (term_equal (car xa) (car ya)) (return nil))
         (setq xa (cdr xa)  ya (cdr ya))
       )
       ))
  ))))

; i is a depth
(defun apply_rule (i r x)
  (let ((mt (term_match (rule_lhs r) x)))
    (unless (eq 'fail mt)
    (if (rule_cond r)
        (let ((res (eval_cond_direct (rule_cond r) mt)))
        (if (eq 'fail res)
          (break "apply_rule: complex condition")
        (when res
          (setq rule_successful t)
          (ens_inc_repl current_ensemble)
          (incf replacements)
          (add_repl x r i (rule_inst r x mt)))))
      (progn
        (setq rule_successful t)
        (ens_inc_repl current_ensemble)
        (incf replacements)
        (add_repl x r i (rule_inst r x mt)))
    )
  )))

;-------------------------------------------------------------------------
(defun apply_rule_all (i r x)
  (unless (or (term_is_var x) (term_is_const x))
    (apply_rule i r x)
    (dolist (e (term_args x))
      (unless (or (is_root e) ;@@ cut off rewriting at transition @@new
                  (not (occurs_in_ensemble e current_ensemble)))
      (apply_rule_all (+ i 1) r e)))
  ))

;-------------------------------------------------------------------------
(defun try_rule (inf rul)
  (setq rule_successful nil)
  (setq forced_relocation nil)
  (dolist (r (status_roots inf))
    (apply_rule_all 0 rul r))
  (or rule_successful forced_relocation)
  )

(defun activate_ensemble (e)
  (let ((current_ensemble e))
  (let ((inf (status_ensemble e)))
  (unless (null (status_roots inf)) ;to end
  (when (null (status_partitions inf))
    (set_status_partitions inf all_partitions)
    (set_status_rules inf (car all_partitions))
    (set_status_all_rules inf (car all_partitions)))
  (let ((init_rules (if (status_rules inf) (status_rules inf)
                        (status_all_rules inf)))
        (init_partitions (status_partitions inf)))
  (let ((rs init_rules)
        (ps init_partitions))
    (block outer
      (loop
       (when (try_rule inf (car rs))
         (set_status_rules inf (cdr rs))
         (return-from outer nil))
       (setq rs (cdr rs))
       (when (null rs) (return)))
      (setq rs (status_all_rules inf))
      (loop
       (when (or (null rs) (eq rs init_rules)) (return))
       (when (try_rule inf (car rs))
         (set_status_rules inf (cdr rs))
         (return-from outer nil))
       (setq rs (cdr rs)))
      (ens_inc_switch current_ensemble) ;@@note only one switch
      (loop
       (when (null ps) (return))
       (setq rs (car ps))
       ; (ens_inc_switch current_ensemble) ;@@todo only count final switch?
       (loop
        (when (try_rule inf (car rs))
          (set_status_rules inf (cdr rs))
          (set_status_all_rules inf (car ps))
          (return-from outer nil))
        (setq rs (cdr rs))
        (when (null rs) (return)))
       (setq ps (cdr ps)))
      (setq ps all_partitions)
      (loop
       (setq rs (car ps))
       ; (ens_inc_switch current_ensemble) ;@@note only one switch
       (loop
        (when (and (eq rs init_rules) (eq ps init_partitions))
          (return-from outer nil))
        (when (try_rule inf (car rs))
          (set_status_rules inf (cdr rs))
          (set_status_all_rules inf (car ps))
          (return-from outer nil))
        (setq rs (cdr rs))
        (when (null rs) (return)))
       (setq ps (cdr ps))
       (when (null ps) (return)))
    )
  ))
  (ens_full_strategy current_ensemble)
  ))))

;-------------------------------------------------------------------------
(defun status_make (e)
  (list e nil nil nil nil nil)
  )

(defun status_ensemble (e) (aref ensemble_status e))

(defun status_roots (e) (nth 1 e))
(defun status_rules (e) (nth 2 e))
(defun status_all_rules (e) (nth 3 e))
(defun status_partitions (e) (nth 4 e))
(defun status_size (e) (nth 5 e))

(defun set_status_roots (e x) (setf (nth 1 e) x))
(defun set_status_rules (e x) (setf (nth 2 e) x))
(defun set_status_all_rules (e x) (setf (nth 3 e) x))
(defun set_status_partitions (e x) (setf (nth 4 e) x))
(defun set_status_size (e x) (setf (nth 5 e) x))

;-------------------------------------------------------------------------
(defun stats_make ()
  (list
   (stats0_init ens_stats_param)
   (stats0_init ens_stats_param)
```

```
   (stats0_init ens_stats_param)
   (stats0_init ens_stats_param)
   (stats0_init ens_stats_param)
   (stats0_init ens_stats_param)
   (stats0_init ens_stats_param)
   (stats0_init ens_stats_param)
   nil)
  )

(defun stats_ensemble (e) (aref ensemble_stats e))

(defun stats_switch (x) (car x))
(defun stats_reloc (x) (cadr x))
(defun stats_reloc_far (x) (caddr x))
(defun stats_push (x) (cadddr x))
(defun stats_repl (x) (nth 4 x))
(defun stats_size (x) (nth 5 x))
(defun stats_input (x) (nth 6 x))
(defun stats_output (x) (nth 7 x))
(defun stats_used (x) (nth 8 x))

(defun stats_inc_switch (x) (inc_val (car x)))
(defun stats_inc_reloc (x) (inc_val (cadr x)))
(defun stats_inc_reloc_far (x) (inc_val (caddr x)))
(defun stats_inc_push (x) (inc_val (cadddr x)))
(defun stats_inc_repl (x) (inc_val (nth 4 x)))
(defun stats_set_used (x) (setf (nth 8 x) t))

(defun stats_add_input (x val) (add_val (nth 6 x) val))
(defun stats_add_output (x val) (add_val (nth 7 x) val))

(defun ens_inc_switch (e) (stats_inc_switch (aref ensemble_stats e)))
(defun ens_inc_reloc (e) (stats_inc_reloc (aref ensemble_stats e)))
(defun ens_inc_reloc_far (e) (stats_inc_reloc_far (aref ensemble_stats e)))
(defun ens_inc_push (e) (stats_inc_push (aref ensemble_stats e)))
(defun ens_inc_repl (e) (stats_inc_repl (aref ensemble_stats e)))
(defun ens_set_used (e) (stats_set_used (aref ensemble_stats e)))

(defun ens_add_input (e val) (stats_add_input (aref ensemble_stats e) val))
(defun ens_add_output (e val) (stats_add_output (aref ensemble_stats e) val))

(defmacro stats0_cur (x) `(nth 1 ,x))

(defun ens_add_size (e)
  (let ((stts (aref ensemble_stats e)))
  (let ((siz (stats_size stts)))
  (setf (stats0_cur siz) (ens_size e))
  (new_group siz))))

(defun ens_stats_new_group (e)
  (let ((stts (aref ensemble_stats e)))
  (new_group (nth 0 stts))
  (new_group (nth 1 stts))
  (new_group (nth 2 stts))
  (new_group (nth 3 stts))
  (new_group (nth 4 stts))
  (let ((siz (state_size stts)))
    (setf (stats0_cur siz) (ens_size e))
    (new_group siz))
  (new_group (nth 6 stts))
  (new_group (nth 7 stts))
  ))

;-----------------------------------------------------------------------
(defun cluster_inc_reloc (i j)
  (inc_val (aref cluster_stats i j)))

;-----------------------------------------------------------------------
(defun cluster_ensemble_inc_reloc (i j)
  (let ((ic (aref ensemble_cluster i)) (jc (aref ensemble_cluster j)))
  (if (= ic jc)
      (progn
        (when (null (aref cluster_ensemble_stats ic))
          (setf (aref cluster_ensemble_state ic)
              (make_2d_array_stats cluster_size cluster_size
                                   clust_ens_stats_param)))
        (inc_val (aref (aref cluster_ensemble_stats ic)
                   (cluster_ensemble_number i)
                   (cluster_ensemble_number j)))
        )
    (cluster_inc_reloc ic jc))
  ))

;-----------------------------------------------------------------------
(defun cluster_ensemble_inc_push (i j)
  (let ((ic (aref ensemble_cluster i)) (jc (aref ensemble_cluster j)))
  (if (= ic jc)
      (progn
        (when (null (aref cluster_ensem`'e_stats2 ic))
          (setf (aref cluster_ensemble_   ts2 ic)
              (make_2d_array_stats clust.  size cluster_size
                                   clust_ens_stats_param)))
        (inc_val (aref (aref cluster_ensemble_stats2 ic)
                   (cluster_ensemble_number i)
                   (cluster_ensemble_number j)))
        )
    (inc_val (aref cluster_stats2 ic jc)))
  ))

;-----------------------------------------------------------------------
; only push if new
(defun is_new (x)
  (assoc 'new_where (term_an x)))

(defun move_node (u e)
  (let ((res (assoc 'new_where (term_an u))))
  (when res
    (setf (car res) 'where)
    (setf (cadr res) (cons e clock))
  )))

;-----------------------------------------------------------------------
(defun update_new_loc (x e)
  (if (term_is_var x) x
  (if (atom x) x
  (if (term_is_bi x) x
    (when (is_new x)
      (move_node x e)
      (dolist (y (term_args x))
        (update_new_loc y e)))
  ))))

;-----------------------------------------------------------------------
(defun init ()
  (setq ensemble_cluster (make-array (list rrm_size)))
  (dotimes (i rrm_size)
    (setf (aref ensemble_cluster i) (rrm_cluster i)))

  (setq ensemble_status (make-array (list rrm_size)))
  (dotimes (i rrm_size)
    (setf (aref ensemble_status i) (status_make i)))

  (setq cluster_stats
        (make_2d_array_stats rrm_cluster_size rrm_cluster_size clust_param))

  (setq cluster_ensemble_state
        (make-array (list rrm_cluster_size)))

  (setq cluster_stats2
        (make_2d_array_stats rrm_cluster_size rrm_cluster_size clust_param))

  (setq cluster_ensemble_stats2
        (make-array (list rrm_cluster_size)))

  (setq ensemble_state (make-array (list rrm_size)))
  (dotimes (i rrm_size)
    (setf (aref ensemble_stats i) (state_make)))

  (setq replacement_state (stats0_init 1))
  (setq replacements 0)
```

```
  (setq input_state (state0_init 10))
  (setq output_state (state0_init 10))

  (setq step_input 0)
  (setq step_output 0)

  (reset_clock)

  (setq action_list nil)
  )

;-------------------------------------------------------------------
(defun setup (file)
  (with-open-file (*standard-input* file :direction :input)
    (setq all_partitions (read)))
  (init)
  )

;-------------------------------------------------------------------
(defun simulation_step ()
  (dotimes (i rrm_size)
    (activate_ensemble i)
  )
  (perform_actions)
  (tick)
  )

;-------------------------------------------------------------------
(defun simul ()
  (loop
   (when (< clock_limit clock)
     (princ "---------------------------------------- ") (terpri)
     (princ "CLOCK LIMIT EXCEEDED") (terpri)
     (return))
   (when verbose
     (when (<= 1 verbose)
       (princ "---------------------------------------- "))
;      (princ "clock ")
     (princ " ")
     (prin1 clock)
     (when (< 60 (col)) (terpri))
     (when (and
            (or (<= 1 verbose)
                (and (< 0 clock)
                     (= 0 (rem clock periodic)))))
     (when (< 0 (col)) (terpri))
     (when show_term
       (princ "term: ") (print_term root) (terpri))
     )
     (when (<= 6 verbose)
     (princ "details: ") (prin root) (terpri)
     (when (<= 8 verbose)
       (dotimes (i rrm_size)
         (let ((inf (status_ensemble i)))
         (when (status_roots inf)
           (princ "-------------------- ") (prin i) (terpri)
           (dolist (x (status_roots inf))
             (princ "root: ") (prin x) (terpri))
           (princ "---------- end") (terpri)
         )
       ))
     ))
     (force-output)
     )
   (setq action_list nil)
   (setq some_relocation nil)
   (simulation_step)
   (when (and (null action_list) (null some_relocation)
      (or (not (fboundp 'check_reduced)) (check_reduced root)))
     (return))
   (when (= 0 (rem clock gc_period)) (perform_gc))
  )
  )
```

```
;-------------------------------------------------------------------
(defun mark_root_node (x)
  (set_an x 'root 'is_root))

(defun is_root (x)
  (eq 'is_root (get_an x 'root)))

(defun mark_node (x)
  (when (is_root x)
  (set_an x 'mark clock))) ;@@ clock versus 'marked

(defun clear_node (x)
;  (set_an x 'mark 0)
  )

(defun is_marked (x)
  (equal clock (get_an x 'mark)))

(defun mark_term (x)
  (unless (or (term_is_var x) (term_is_bi x) (is_marked x))
    (mark_node x)
    (dolist (y (term_args x))
      (mark_term y))
  ))

(defun perform_gc ()
  (when verbose
    (when (<= 1 verbose)
    (princ "----------------------------------------"))
    (princ " GC")
    (when (<= 1 verbose) (terpri))
  )
  (dotimes (i rrm_size) ;@@todo eliminate need for (put in ensemble_add_root)
    (let ((inf (status_ensemble i)))
    (dolist (r (status_roots inf))
      (mark_root_node r)
      (clear_node r)
    )
  ))
  (mark_term root)
  (dotimes (i rrm_size)
    (let ((inf (status_ensemble i)))
    (when (status_roots inf)
      (set_status_roots inf
        (mapcan #'(lambda (n)
            (if (is_marked n)
                (progn
                  (clear_node n)
                  (list n)
                  )
              (progn (clear_node n) nil)
              ))
          (status_roots inf)))
    )
  ))
;  (fix_roots_where root) ;@@todo fix this
  )

(defun fix_roots_where (x)
  (unless (or (term_is_var x) (term_is_bi x))
    (when (is_root x)
    (let ((new
      (mapcan #'(lambda (p)
           (when (member x (status_roots (status_ensemble (car p))))
             (list p))
         )
        (get_an x 'where))))
      (when (null new)
        (princ "#### null where") (terpri)
        (prin1 x) (terpri)
        (prin1 (get_an x 'where)) (terpri)
        (break))
      (if (null new)
          ()
        (set_an x 'where new))
```

```
    ))
    (dolist (y (term_args x))
      (fix_roots_where y))
  ))

;-----------------------------------------------------------------------
(defun init_term (e x)
  (let ((verbose nil) (current_ensemble 0))
  (build_term t e 0 x)
  ))

;-----------------------------------------------------------------------
(defun ens_size (e)
  (let ((inf (status_ensemble e)))
  (let ((x (status_size inf)))
  (if (and x (= clock (car x))) (cdr x)
    (progn
      (when (null x)
        (setq x (cons nil nil))
        (set_status_size inf x))
      (setf (cdr x) (compute_ens_size e))
      (setf (car x) clock)
      (cdr x)
    ))
  )))

(defun ens_size_recompute (e)
  (let ((inf (status_ensemble e)))
  (let ((x (status_size inf)))
    (when (null x)
      (setq x (cons nil nil))
      (set_status_size inf x))
    (setf (cdr x) (compute_ens_size e))
    (setf (car x) clock)
    (cdr x)
  )))

(defun term_top_size (x)
  (if (or (term_is_var x) (term_is_const x)) 1
    (let ((res 1)) ;count one for the top node
      (dolist (e (term_args x))
        (unless (is_root e)
          (setq res (+ res (term_top_size e))))
      )
      res
    )
  ))

(defun term_size (x)
  (if (or (term_is_var x) (term_is_const x)) 1
    (let ((res 1)) ;count one for the top node
      (dolist (e (term_args x))
          (setq res (+ res (term_size e)))
      )
      res
    )
  ))

(defun compute_ens_size (e)
  (let ((res 0)
        (inf (status_ensemble e)))
    (dolist (r (status_roots inf))
      (setq res (+ res (term_top_size r)))
    )
    res
  ))

;-----------------------------------------------------------------------
(unless (fboundp 'filecol) (load "filecol"))

(defun col () (filecol *standard-output*))

(defun prin (x)
  (when (< 60 (col)) (terpri) (princ "  "))
  (if (atom x) (prin1 x)
    (progn
      (princ "(")
      (prin (car x))
      (setq x (cdr x))
      (loop
       (when (atom x) (when x (princ " . ") (prin x)) (return))
       (princ " ")
       (prin (car x))
       (setq x (cdr x))
      )
      (when (< 60 (col)) (terpri) (princ "  "))
      (princ ")")
    )))

(defvar to-file t)
(defmacro to-file (file &rest forms)
  (if to-file
  `(with-open-file (*standard-output* ,file :direction :output)
     (let ((*trace-output* *standard-output*)
           (currentfile *standard-output*))
       ,@forms))
  `(progn ,@forms)
  ))
;-----------------------------------------------------------------------
;statistics stuff
; lisp object that flexibly and extensibly keeps record of statistics
; -- add_val, new_group

; stats:
;       current group
;               number, sum, min, max, sum-of-sq, distribution(/k)
; ... keep adaptive distribution where know how many values excluded (came
;    before the last adaptation)?

; a tag indicating the treatment (for extensibility)
(defun stats_kind (x) (car x))
; stats0,stats1,...

(defun inc_val (sts)
  (if (eq 'stats0 (stats_kind sts))
      (stats0_inc_val sts)
    (break "inc_val: unknown kind")
  ))

(defun add_val (sts val)
  (if (eq 'stats0 (stats_kind sts))
      (stats0_add_val sts val)
    (break "add_val: unknown kind")
  ))

(defun insert_val (sts v)
  (if (eq 'stats0 (stats_kind sts))
      (stats0_insert_val sts v)
    (break "insert_val: unknown kind")
  ))

(defun new_group (sts)
  (if (eq 'stats0 (stats_kind sts))
      (stats0_new_group sts)
    (break "new_group: unknown kind")
  ))

(defun compute_derived (sts)
  (if (eq 'stats0 (stats_kind sts))
      (stats0_compute_derived sts)
    (break "compute_derived: unknown kind")
  ))

;stats0 format
;  (stats0 (current_stats) (global_stats) dist-param)
; current_stats
;  number sum min max sum-of-sq distribution
; global_stats
;  number sum min max sum-of-sq distribution
```

```
(defmacro stats0_cur (x) `(nth 1 ,x))
(defmacro stats0_glbl (x) `(nth 2 ,x))
(defmacro stats0_distr_param (x) `(nth 3 ,x))

(defmacro stats0_num (x) `(nth 0 ,x))
(defmacro stats0_sum (x) `(nth 1 ,x))
(defmacro stats0_min (x) `(nth 2 ,x))
(defmacro stats0_max (x) `(nth 3 ,x))
(defmacro stats0_sum_sq (x) `(nth 4 ,x))
(defmacro stats0_distr (x) `(nth 5 ,x))
(defmacro stats0_av (x) `(nth 6 ,x))
(defmacro stats0_dev (x) `(nth 7 ,x))

(defun coerce-float (x) (coerce x 'long-float))

(defvar count_stats 0)

(defun stats0_compute_derived (stts)
  (incf count_stats)
  (let ((ss (stats0_glbl stts)))
  (let ((n (stats0_num ss))
        (sum (stats0_sum ss))
        (sum_sq (stats0_sum_sq ss)))
  (if (= 0 n)
    (progn
      (setf (stats0_av ss) 0.0)
      (setf (stats0_dev ss) 0.0))
  (let ((av (coerce-float (/ sum n))))
  (setf (stats0_av ss) av)
  (setf (stats0_dev ss)
      (sqrt (- (coerce-float (/ sum_sq n)) (* av av))))))

  )))))

(defun stats0_create ()
  (list 0 0 most-positive-fixnum most-negative-fixnum 0 nil nil nil)
  )

(defun stats0_init (p)
  (list 'stats0 0 (stats0_create) p)
  )

(defun stats0_inc_val (sts)
  (incf (stats0_cur sts))
  )

(defun stats0_add_val (sts val)
  (incf (stats0_cur sts) val)
  )

(defun stats0_insert_val (sts v)
  (setf (stats0_cur sts) v)
  )

(defun stats0_new_group (sts)
  (let ((n (stats0_cur sts))
        (ss (stats0_glbl sts))
        (dp (stats0_distr_param sts)))
  (incf (stats0_num ss))
  (setf (stats0_sum ss) (+ (stats0_sum ss) n))
  (incf (stats0_sum ss) n)
  (incf (stats0_sum_sq ss) (* n n))

  (let ((val (stats0_min ss)))
  (when (or (null val) (< n val))
    (setf (stats0_min ss) n)))

  (let ((val (stats0_max ss)))
  (when (or (null val) (< val n))
    (setf (stats0_max ss) n)))

  (setf (stats0_distr ss) (add_distr n 1 dp (stats0_distr ss)))

  (setf (stats0_cur sts) 0) ;reset counter
  ))
```

```
(defun add_distr (n c p d)
  (let ((v (* p (truncate n p))))
  (add_distr1 v c d)
  ))

(defun add_distr1 (v c d)
  (let ((l d))
  (loop
   (when (null l) (return (insert_distr v c d)))
   (when (= v (caar l)) (incf (cdar l) c) (return d))
   (setq l (cdr l))
  )
  ))

(defun insert_distr (n c d)
  (let ((pr (cons n c)))
  (if (null d) (cons pr nil)
  (if (< n (caar d)) (cons pr d)
  (let ((1st d) (cur (cdr d)))
    (loop
     (when (null cur) (setf (cdr lst) (cons pr nil)) (return))
     (when (< n (caar cur)) (setf (cdr lst) (cons pr cur)) (return))
     (setq lst cur  cur (cdr cur))
    )
    d
  )))))

(defun merge_state (ss gs)
  (incf (stats0_num gs) (stats0_num ss))
  (incf (stats0_sum gs) (stats0_sum ss))
  (incf (stats0_sum_sq gs) (stats0_sum_sq ss))

  (let ((vals (stats0_min ss))
        (val (stats0_min gs)))
  (when (or (null val) (< vals val))
    (setf (stats0_min gs) vals)))

  (let ((vals (stats0_max ss))
        (val (stats0_max gs)))
  (when (or (null val) (< val vals))
    (setf (stats0_max gs) vals)))

  (setf (stats0_distr gs) (merge_distr (stats0_distr ss) (stats0_distr gs)))
  )

(defun merge_distr (d1 d2)
  (dolist (de d1)
    (setq d2 (add_distr1 (car de) (cdr de) d2)))
  d2
  )

;-----------------------------------------------------------------------
(defun make_2d_array_stats (m n p)
  (let ((res (make-array (list m n))))
  (dotimes (i m)
    (dotimes (j n)
      (setf (aref res i j)
          (stats0_init p))))
  res
  ))

;-------------------------------------------- ------------------------
(defun all_new_group ()
  (insert_val replacement_stats replacements)
  (new_group replacement_stats)
  (setq replacements 0)
  (global_io_new_group)
  (dotimes (e rrm_size)
    (ens_stats_new_group e))
  (dotimes (i rrm_cluster_size)
    (dotimes (j rrm_cluster_size)
      (new_group (aref cluster_stats i j))
      (new_group (aref cluster_stats2 i j))
      )
```

```
    (let ((val (aref cluster_ensemble_stats i)))
    (when val
      (dotimes (a cluster_size)
        (dotimes (b cluster_size)
          (new_group (aref val a b))))))
    (let ((val (aref cluster_ensemble_stats2 i)))
    (when val
      (dotimes (a cluster_size)
        (dotimes (b cluster_size)
          (new_group (aref val a b))))))
    )
  )

(defun all_stats_apply (fn)
  (dotimes (e rrm_size)
    (let ((stts (aref ensemble_stats e)))
    (dotimes (i 5)
      (funcall fn (nth i stts)))))
  (dotimes (i rrm_cluster_size)
    (dotimes (j rrm_cluster_size)
      (funcall fn (aref cluster_stats i j))
      (funcall fn (aref cluster_stats2 i j))
      )
    (let ((val (aref cluster_ensemble_stats i)))
    (when val
      (dotimes (a cluster_size)
        (dotimes (b cluster_size)
          (funcall fn (aref val a b))))))
    (let ((val (aref cluster_ensemble_stats2 i)))
    (when val
      (dotimes (a cluster_size)
        (dotimes (b cluster_size)
          (funcall fn (aref val a b))))))
    )
  )
```

```
(defun activate_ensemble (e)
  (let ((current_ensemble e))
  (let ((inf (status_ensemble e)))
  (unless (null (status_roots inf)) ;to end
  (when (null (status_partitions inf))
    (set_status_partitions inf all_partitions)
    (set_status_rules inf (car all_partitions))
    (set_status_all_rules inf (car all_partitions)))
  (let ((init_rules (if (status_rules inf) (status_rules inf)
                        (status_all_rules inf)))
        (init_partitions (status_partitions inf)))
  (let ((rs init_rules)
        (ps init_partitions))
    (block outer
      (loop
       (prinl e) (princ " -->1 ") (prinl (car rs)) (terpri)
       (when (try_rule inf (car rs))
       (prinl e) (princ " !!!!-->1 ") (prinl (car rs)) (terpri)
         (set_status_rules inf (cdr rs))
         (return-from outer nil))
       (setq rs (cdr rs))
       (when (null rs) (return)))
      (setq rs (status_all_rules inf))
      (loop
       (when (or (null rs) (eq rs init_rules)) (return))
       (prinl e) (princ " -->2 ") (prinl (car rs)) (terpri)
       (when (try_rule inf (car rs))
       (prinl e) (princ " !!!!-->2 ") (prinl (car rs)) (terpri)
         (set_status_rules inf (cdr rs))
         (return-from outer nil))
       (setq rs (cdr rs)))
      (ens_inc_switch current_ensemble) ;@@note only one switch
      (loop
       (when (null ps) (return))
       (setq rs (car ps))
       ; (ens_inc_switch current_ensemble) ;@@todo only count final switch?
       (loop
       (prinl e) (princ " -->3 ") (prinl (car rs)) (terpri)
        (when (try_rule inf (car rs))
       (prinl e) (princ " !!!!-->3 ") (prinl (car rs)) (terpri)
          (set_status_rules inf (cdr rs))
          (set_status_all_rules inf (car ps))
          (return-from outer nil))
        (setq rs (cdr rs))
        (when (null rs) (return)))
       (setq ps (cdr ps)))
      (setq ps all_partitions)
      (loop
       (setq rs (car ps))
       ; (ens_inc_switch current_ensemble) ;@@note only one switch
       (loop
        (when (and (eq rs init_rules) (eq ps init_partitions))
          (return-from outer nil))
       (prinl e) (princ " -->4 ") (prinl (car rs)) (terpri)
        (when (try_rule inf (car rs))
       (prinl e) (princ " !!!!-->4 ") (prinl (car rs)) (terpri)
          (set_status_rules inf (cdr rs))
          (set_status_all_rules inf (car ps))
          (return-from outer nil))
        (setq rs (cdr rs))
        (when (null rs) (return)))
       (setq ps (cdr ps))
       (when (null ps) (return)))
    )
  ))
  (ens_full_strategy current_ensemble)
  ))))
```

```
(defvar alloc_range 100)
(defvar alloc_cut 2)
(defvar alloc_thres 100)
(defvar alloc_num 0)
(defvar alloc_full 0)
(defvar alloc_totally_full 0)
(defvar alloc_totally_full_random nil)

; right 0
; up 1
; left 2
; down 3
(defun next_dir (x) (rem (+ x 1) 4))

(defun rrm_ens_dir (e x)
  (case x
    (0 (rrm_right e))
    (1 (rrm_up e))
    (2 (rrm_left e))
    (3 (rrm_down e))
  ))

(defun alloc_ensemble (e)
  (alloc_ens e t))

(defun alloc_ens (e flag)
  ; when flag try allocating in other clusters, otherwise fail (nil value)
  (incf alloc_num)
  (let ((ch (random alloc_range)))
  (let ((dir
         (if (< ch alloc_cut)
           (if (evenp ch) 1 2)
           (if (evenp ch) 0 3))))
  (let ((cdir dir))
  (loop
   (let ((ce (rrm_ens_dir e cdir)))
   (let ((size (ens_size ce)))
     (when (< size alloc_thres) (return ce))
     (setq cdir (next_dir cdir))
     (when (= dir cdir)
       (when (<= 5 verbose)
         (princ "### alloc full ens: ") (princ1 e) (terpri))
       (incf alloc_full)
       (let ((c (aref ensemble_cluster e)) (res nil))
       (block search
         (dotimes (i cluster_height)
           (dotimes (j cluster_width)
             (let ((ce (cluster_elt c i j)))
             (let ((size (ens_size ce)))
             (when (< size alloc_thres)
               (setq res ce)
               (return-from search nil))))))
         ;(setq res (rrm_ens_dir e dir)) ; if try all use first
         (when (<= 5 verbose)
           (princ "### alloc totally full ens ") (prin1 e)
           (princ " in ") (princ1 c) (terpri))
         (incf alloc_totally_full)
         (when flag
           (unless alloc_totally_full_random
             (dotimes (i 4)
               (setq res
                   (alloc_ens
                    (cluster_elt
                     (case i
                       (0 (rrm_cluster_right c))
                       (1 (rrm_cluster_down c))
                       (2 (rrm_cluster_left c))
                       (3 (rrm_cluster_up c)))
                     (random cluster_height)
                     (random cluster_width))
                    nil)) ; fail on totally full
               (when res (return)))
             ) ; unless
           (unless res
             (setq res (random rrm_size))) ; choose a random ensemble
           ); when
         )
       (return res)))
  )))
  ))))

(defun rrm_cluster_left (e)
  (let ((col (rem e rrm_cluster_width)))
    (+ e (- col) (if (= col 0) (- rrm_cluster_width 1) (- col 1)))))

(defun rrm_cluster_right (e)
  (let ((col (rem e rrm_cluster_width)))
    (+ e (- col)
       (if (= col (- rrm_cluster_width 1)) 0 (+ col 1)))))

(defun rrm_cluster_up (e)
  (rem (+ e rrm_cluster_size (- rrm_cluster_width)) rrm_cluster_size))

(defun rrm_cluster_down (e)
  (rem (+ e rrm_cluster_width) rrm_cluster_size))
```

```
; updated now that stats0 used for
;       stats in ensemble_stats (func stats_ensemble)
;       cluster stats for ensembles in cluster_ensemble_stats
;               and cluster_ensemble_stats2
;       cluster_stats
;       cluster_stats2

(defun stats0_val (x) (stats0_sum (stats0_glbl x)))

(defun shm ()
  (princ
"------------------------------------------------------------------------")
  (terpri))

(defun v ()
  (setq count_stats 0)
  (all_state_apply #'stats0_compute_derived)
  (princ "number of stats: ") (prin1 count_stats) (terpri)

  (shm)
  (princ "root: ") (print_term root) (terpri)
;  (princ "details: ") (prin root) (terpri)

  (shm)
  (princ "roots") (terpri)
  (ens_rts)

  (shm)
  (princ "size") (terpri)
  (ens_siz)

  (shm)
  (princ "maximum size") (terpri)
  (ens_max_siz)

  (shm)
  (princ "cluster matrix") (terpri)
  (clust_mat)

  (when (boundp 'alloc_num)
  (shm)
  (princ "alloc num: ") (prin1 alloc_num) (terpri)
  (princ "alloc full: ") (prin1 alloc_full) (terpri)
  (princ "alloc totally full: ") (prin1 alloc_totally_full) (terpri)
  )

  (when (boundp 'conflict_near)
  (shm)
  (princ "near communication conflicts: ") (prin1 conflict_near) (terpri)
  (princ "far communication conflicts: ") (prin1 conflict_far) (terpri)
  )
  (when (boundp 'conflict_reloc_size)
  (princ "reloc size conflicts: ") (prin1 conflict_reloc_size) (terpri)
  )
  (when (boundp 'ens_full_push)
  (shm)
  (princ "ensemble full pushes: ") (prin1 ens_full_push) (terpri)
  (princ "ensemble full pushes with single root: ")
      (prin1 ens_full_push_single) (terpri)
  )

  (when (boundp 'replacement_stats)
  (shm)
  (compute_derived replacement_stats)
  (princ "replacement per step stats") (terpri)
  (stats0_show_stats2 replacement_stats 0)
  )

  (ensstats)

  (shm)
  (shc)

  (shm)
  (shc2)

  (unless (fboundp 'ensiostats) (load "a2"))
  (ensiostats)
  )

(defun clust_mat ()
  (princ "     ")
  (dotimes (i rrm_cluster_size) (format t "~4d" i)) (terpri)
  (dotimes (i rrm_cluster_size)
    (format t "~2d: |" i)
    (dotimes (j rrm_cluster_size)
      (if (= i j) (if (not (= 0 (stats0_val (aref cluster_stats i j))))
                      (princ " ###")
                      (princ " ---"))
      (let ((v (stats0_val (aref cluster_stats  i j))))
      (format t "~4d" v)
    )))
    (princ " |") (terpri)
  )
  )

(defun ens_rts ()
  (princ "      ")
  (dotimes (i rrm_width) (format t "~4d" i)) (terpri)
  (dotimes (i rrm_height)
    (format t "~3d: |" i)
    (dotimes (j rrm_width)
      (let ((e (rrm_elt i j)))
      (let ((rs (status_roots (status_ensemble e))))
      (if rs
          (format t "~4d" (length rs))
      (if (stats_used (state_ensemble e)) (princ "   o")
        (princ "   ."))
    ))))
    (princ " |") (terpri)
  )
  )

(defun ens_siz ()
  (princ "      ")
  (dotimes (i rrm_width) (format t "~4d" i)) (terpri)
  (dotimes (i rrm_height)
    (format t "~3d: |" i)
    (dotimes (j rrm_width)
      (let ((e (rrm_elt i j)))
      (let ((sz (ens_size e)))
      (if (< 0 sz)
          (format t "~4d" sz)
      (if (stats_used (state_ensemble e)) (princ "   o")
        (princ "   .")))
    )))
    (princ " |") (terpri)
  )
  )

(defun ens_max_siz ()
  (princ "      ")
  (dotimes (i rrm_width) (format t "~4d" i)) (terpri)
  (dotimes (i rrm_height)
    (format t "~3d: |" i)
    (dotimes (j rrm_width)
      (let ((e (rrm_elt i j)))
      (let ((ss (stats_ensemble e)))
      (let ((maxsiz (stats0_max (stats0_glbl (stats_size ss)))))
      (if (< 0 maxsiz)
          (format t "~4d" maxsiz)
      (if (stats_used ss) (princ "   o")
        (princ "   .")))
    ))))
    (princ " |") (terpri)
  )
  )

(defun ensstats ()
  (shm)
  (princ "global ensemble stats") (terpri)
```

```
  (let ((num 0)
        (used 0)
        (sw 0)
        (rel 0)
        (rel_far 0)
        (push 0)
        (repl 0)
        (maxsize 0)
        (maxsize0 0)
        (maxsizesum 0)
        (size 0)
        (none (stats_make)))
  (dotimes (i rrm_size)
    (let ((ss (stats_ensemble i)))
    (unless (equal none ss)
      (setq num (+ num 1))
      (when (stats_used ss) (setq used (+ used 1)))
      (setq sw (+ sw (stats0_val (stats_switch ss))))
      (setq rel (+ rel (stats0_val (stats_reloc ss))))
      (setq rel_far (+ rel_far (stats0_val (stats_reloc_far ss))))
      (setq repl (+ repl (stats0_val (stats_repl ss))))
      (setq push (+ push (stats0_val (stats_push ss))))
      (let ((mx (stats0_max (stats0_glbl (stats_size ss)))))
        (if (= 0 mx) (incf maxsize0)
          (progn
            (when (< maxsize mx) (setq maxsize mx))
            (setq maxsizesum (+ maxsizesum mx)))))
      (setq size (+ size (ens_size i))) ;@@
    )))
  (princ "clock: ") (prin1 clock) (terpri)
  (princ "num: ") (prin1 num) (terpri)
  (princ "used: ") (prin1 used) (terpri)
  (princ "sw: ") (prin1 sw) (terpri)
  (princ "rel: ") (prin1 rel) (terpri)
  (princ "rel_far: ") (prin1 rel_far) (terpri)
  (princ "push: ") (prin1 push) (terpri)
  (princ "repl: ") (prin1 repl) (terpri)
  (princ "max size: ") (prin1 maxsize) (terpri)
  (princ "num max size = 0: ") (prin1 maxsize0) (terpri)
  (princ "av max size: ") (prin1 (coerce (/ maxsizesum num) 'float)) (terpri)
  (princ "av max size excluding 0: ")
    (prin1 (coerce (/ maxsizesum (- num maxsize0)) 'float)) (terpri)
  (princ "size: ") (prin1 size) (terpri)
;  (princ "-------------------- av over ensembles") (terpri)
;  (princ "av used: ") (prin1 (coerce (/ used num) 'float)) (terpri)
;  (princ "av sw: ") (prin1 (coerce (/ sw num) 'float)) (terpri)
;  (princ "av rel: ") (prin1 (coerce (/ rel num) 'float)) (terpri)
;  (princ "av rel_far: ") (prin1 (coerce (/ rel_far num) 'float)) (terpri)
;  (princ "av push: ") (prin1 (coerce (/ push num) 'float)) (terpri)
;  (princ "av repl: ") (prin1 (coerce (/ repl num) 'float)) (terpri)
  (if (= 0 num) (princ "num = 0")
    (progn
    (princ "av size: ") (prin1 (coerce (/ size num) 'float)) (terpri)))
  (princ "-------------------- av per tick") (terpri)
  (unless (= 0 clock)
  (princ "av used: ") (prin1 (coerce (/ used clock) 'float)) (terpri)
  (princ "av sw: ") (prin1 (coerce (/ sw clock) 'float)) (terpri)
  (princ "av rel: ") (prin1 (coerce (/ rel clock) 'float)) (terpri)
  (princ "av rel_far: ") (prin1 (coerce (/ rel_far clock) 'float)) (terpri)
  (princ "av push: ") (prin1 (coerce (/ push clock) 'float)) (terpri)
  (princ "av repl: ") (prin1 (coerce (/ repl clock) 'float)) (terpri))
  (princ "-------------------- av per ensemble per tick") (terpri)
  (let ((cnt (* num clock)))
  (unless (= 0 cnt)
  (princ "av used: ") (prin1 (coerce (/ used cnt) 'float)) (terpri)
  (princ "av sw: ") (prin1 (coerce (/ sw cnt) 'float)) (terpri)
  (princ "av rel: ") (prin1 (coerce (/ rel cnt) 'float)) (terpri)
  (princ "av rel_far: ") (prin1 (coerce (/ rel_far cnt) 'float)) (terpri)
  (princ "av push: ") (prin1 (coerce (/ push cnt) 'float)) (terpri)
  (princ "av repl: ") (prin1 (coerce (/ repl cnt) 'float)) (terpri))
  )
  )
  )

(defun dmp ()
  (shm)
  (princ "detailed ensemble info") (terpri)
  (let ((none (stats_make)))
  (dotimes (i rrm_size)
    (let ((ss (stats_ensemble i)))
    (unless (equal none ss)
      (princ "-------------------- ens: ") (prin1 i) (terpri)
      (princ "roots: ") (prin1 (length (status_roots (status_ensemble i))))
        (terpri)
      (princ "switch: ") (prin1 (stats_switch ss)) (terpri)
      (princ "reloc: ") (prin1 (stats_reloc ss)) (terpri)
      (princ "reloc_far: ") (prin1 (stats_reloc_far ss)) (terpri)
      (princ "push: ") (prin1 (stats_push ss)) (terpri)
      (princ "repl: ") (prin1 (stats_repl ss)) (terpri)
      (princ "size: ") (prin1 (ens_size i)) (terpri) ;@@
    ))
  ))
  (shm)
  (princ "ensemble roots") (terpri)
  (dotimes (i rrm_size)
    (let ((rs (status_roots (status_ensemble i))))
    (when rs
      (princ "-------------------- ens ") (prin1 i) (terpri)
      (dolist (r rs)
        (prin r) (terpri)
        (princ "###") (terpri))
    )
  )))

(defun ensnum (name)
  (shm)
  (prin1 name) (terpri)
  (princ "     ")
  (dotimes (i rrm_width) (format t "~4d" i)) (terpri)
  (dotimes (i rrm_height)
    (format t "~3d: |" i)
    (dotimes (j rrm_width)
      (let ((e (rrm_elt i j)))
      (let ((res (funcall name (stats_ensemble e))))
        (format t "~4d" res)
      )))
    (princ " |") (terpri)
  )
  )

(defun ens_name (name)
  (shm)
  (prin1 name) (terpri)
  (princ "     ")
  (dotimes (i rrm_width) (format t "~2d" i)) (terpri)
  (dotimes (i rrm_height)
    (format t "~3d: |" i)
    (dotimes (j rrm_width)
      (let ((e (rrm_elt i j)))
      (let ((res (funcall name (stats_ensemble e))))
        (if (numberp res)
            (if (= 0 res) (princ " .")
            (if (and (<= 0 res) (<= res 9)) (progn (princ " ") (prin1 res))
            (case (truncate res 100)
              (0 (princ " a"))
              (1 (princ " b"))
              (2 (princ " c"))
              (3 (princ " d"))
              (4 (princ " e"))
              (otherwise (princ " O")))))
          (format t "~2a" res))
    )))
    (princ " |") (terpri)
  )
  )

(defun clust_ens_stats (e) (clust_stats (aref ensemble_cluster e)))

(defun clust_stats (ec)
  (let ((mat (aref cluster_ensemble_state ec)))
```

```
  (when mat
  (princ "     ")
  (dotimes (i cluster_size) (format t "~4d" i)) (terpri)
  (dotimes (i cluster_size)
    (format t "~2d: |" i)
    (dotimes (j cluster_size)
      (let ((v (aref mat  i j)))
      (format t "~4d" v)
    ))
    (princ " |") (terpri)
  )
  )))

(defun clust_ens_stats_2d (e) (clust_stats_2d (aref ensemble_cluster e)))

(defun clust_stats_2d (ec)
  (let ((mat (aref cluster_ensemble_stats ec)))
  (when mat
  (princ "     ")
  (dotimes (i cluster_width) (format t "~4d" i)) (terpri)
  (dotimes (i cluster_height)
    (format t "~2d: |" i)
    (dotimes (j cluster_width)
      (let ((v 0) (wh (ensemble_number i j)))
        (dotimes (k cluster_size)
          (setq v (+ v (aref mat wh k))))
        (format t "~4d" v)
    ))
    (princ " |") (terpri)
  )
  )))

(defun clust_ens_stats2 (e) (clust_stats2 (aref ensemble_cluster e)))

(defun clust_stats2 (ec)
  (let ((mat (aref cluster_ensemble_stats2 ec)))
  (when mat
  (princ "     ")
  (dotimes (i cluster_size) (format t "~4d" i)) (terpri)
  (dotimes (i cluster_size)
    (format t "~2d: |" i)
    (dotimes (j cluster_size)
      (let ((v (aref mat  i j)))
      (format t "~4d" v)
    ))
    (princ " |") (terpri)
  )
  )))

(defun clust_ens_stats2_2d (e) (clust_stats2_2d (aref ensemble_cluster e)))

(defun clust_stats2_2d (ec)
  (let ((mat (aref cluster_ensemble_stats2 ec)))
  (when mat
  (princ "     ")
  (dotimes (i cluster_width) (format t "~4d" i)) (terpri)
  (dotimes (i cluster_height)
    (format t "~2d: |" i)
    (dotimes (j cluster_width)
      (let ((v 0) (wh (ensemble_number i j)))
        (dotimes (k cluster_size)
          (setq v (+ v (aref mat wh k))))
        (format t "~4d" v)
    ))
    (princ " |") (terpri)
  )
  )))

(defun ens_sketch (e)
  (let ((ss (stats_ensemble e)))
  (if (equal ss (stats_make))
    (progn (princ "none") (terpri))
    (let ((ec (aref ensemble_cluster e)))
      (princ "-------------------- ens: ") (prin1 e)
      (princ " cluster: ") (prin1 ec) (terpri)
      (princ "roots: ") (prin1 (length (status_roots (status_ensemble e))))
      (terpri)
      (princ "switch: ") (prin1 (stats_switch ss)) (terpri)
      (princ "reloc: ") (prin1 (stats_reloc ss)) (terpri)
      (princ "reloc_far: ") (prin1 (stats_reloc_far ss)) (terpri)
      (princ "push: ") (prin1 (stats_push ss)) (terpri)
      (princ "repl: ") (prin1 (stats_repl ss)) (terpri)
      (princ "size: ") (prin1 (stats_size ss)) (terpri)
      (princ "computed size: ") (prin1 (ens_size e)) (terpri) ;@@
      (princ "---------- roots") (terpri)
      (let ((rs (status_roots (status_ensemble e))))
        (dolist (r rs)
          (print_term_top r) (terpri)))
      (let ((en (cluster_ensemble_number e)))
      (let ((elft
             (let ((v (rrm_left e)))
               (if (= ec (aref ensemble_cluster v))
                 (cluster_ensemble_number v)
                 -1)))
            (ergt
             (let ((v (rrm_right e)))
               (if (= ec (aref ensemble_cluster v))
                 (cluster_ensemble_number v)
                 -1)))
            (eup
             (let ((v (rrm_up e)))
               (if (= ec (aref ensemble_cluster v))
                 (cluster_ensemble_number v)
                 -1)))
            (edwn
             (let ((v (rrm_down e)))
               (if (= ec (aref ensemble_cluster v))
                 (cluster_ensemble_number v)
                 -1))))
      (let ((mat (aref cluster_ensemble_stats ec))
            (mat2 (aref cluster_ensemble_stats2 ec)))
      (princ "     reloc     push") (terpri)
      (princ "ens: from   to from   to") (terpri)
      (dotimes (j cluster_size)
        (format t "~3d: ~4d ~4d ~4d ~4d" j
          (if mat (stats0_sum (aref mat en j)) 0)
          (if mat (stats0_sum (aref mat j en)) 0)
          (if mat2 (stats0_sum (aref mat2 en j)) 0)
          (if mat2 (stats0_sum (aref mat2 j en)) 0))
        (when (= j en) (princ " <---"))
        (when (= j elft) (princ " left"))
        (when (= j ergt) (princ " right"))
        (when (= j eup) (princ " up"))
        (when (= j edwn) (princ " down"))
        (terpri)
        ))))
    ))))

(defun print_term_top (x)
  (when (< 60 (col)) (terpri) (princ "  "))
  (if (term_is_var x) (prin1 x)
  (if (or (numberp x) (term_is_bi x)) (prin1 (term_const_val x))
  (progn
    (princ "(")
    (prin1 (term_op x))
    (dolist (e (term_args x))
      (if (is_root e)
        (progn
          (princ " [")
          (prin1 (term_root_loc e))
          (princ "]")
          )
        (progn
          (princ " ")
          (print_term_top e))))
    (when (< 60 (col)) (terpri) (princ "  "))
    (princ ")")
  ))))

(defun cluster_ensemble (i j e)
```

```
  (rrm ensemble i j (truncate e cluster_width) (cluster_col e))
  )

(defun ens_rt (e)
  (princ "ensemble roots") (terpri)
  (let ((rs (status_roots (status_ensemble e))))
    (when rs
      (dolist (r rs)
        (prin r) (terpri)))))

(defun clust_sketch (c)
  (princ "cluster: ") (prin1 c) (terpri)
  (princ "reloc") (terpri)
  (clust_stats c)
  (princ "push") (terpri)
  (clust_stats2 c)
  (princ "reloc from") (terpri)
  (clust_stats_2d c)
  (princ "push from") (terpri)
  (clust_stats2_2d c)
  )

(defvar scan_stats nil)
(defvar scan_zero 0)
(defvar scan_param 1)

(defun shc ()
  (princ "relocation within cluster") (terpri)
  (setq scan_stats (stats0_init scan_param)
        scan_zero 0)
  (dotimes (e rrm_cluster_size)
    (sh e)
    )
  (unless (= 0 (stats0_num (stats0_glbl scan_stats)))
  (show_stats scan_stats scan_zero))
  (setq rel scan_stats relz scan_zero)
  )

(defun show_stats (ss zero)
  (let ((g (stats0_glbl ss)))
  (let ((num (stats0_num g)))
  (princ "number of zero values: ") (prin1 zero) (terpri)
  (setf (stats0_num g) (- num zero))
  (stats0_compute_derived ss)
  (princ "global not zero ") (stats0_show_stats2 ss zero) (terpri)
  (setf (stats0_num g) num)
  (stats0_compute_derived ss)
  (princ "global ") (stats0_show_stats2 ss 0) (terpri)
  (decf count_stats 2)
  )))

(defun sh (e)
  (let ((m (aref cluster_ensemble_stats e)))
  (unless (null m)
    (princ "reloc cluster ") (prin1 e) (terpri)
    (sha m)
    )
  ))

(defun sha (m)
  (unless (null m)
  (progn
  (princ "       ")
  (dotimes (i cluster_size) (format t "~4d" i)) (terpri)
  (dotimes (i cluster_size)
    (format t "~4d: |" i)
    (dotimes (j cluster_size)
      (if (= i j) (princ " ---")
      (let ((e (aref m i j)))
      (let ((val (g (stats0_glbl e))))
        (insert_val scan_stats val) (new_group scan_stats)
        (when (= 0 val) (incf scan_zero))
        (if (= 0 val) (princ "    ")
          (format t " -3d" val))
      ))))
    (princ " |") (terpri)
  )))
  )

(defun g (x)
  (stats0_max x))

(defun shc2 ()
  (princ "pushes within cluster") (terpri)
  (setq scan_stats (stats0_init scan_param)
        scan_zero 0)
  (dotimes (e rrm_cluster_size)
    (sh2 e)
    )
  (unless (= 0 (stats0_num (stats0_glbl scan_stats)))
  (show_stats scan_stats scan_zero)))

(defun sh2 (e)
  (let ((m (aref cluster_ensemble_stats2 e)))
  (unless (null m)
    (princ "push cluster ") (prin1 e) (terpri)
    (sha m)
    )
  ))

(defun stats0_show_stats2 (x zero)
  (let ((ss (stats0_glbl x)))
  (princ "number: ") (prin1 (stats0_num ss)) (terpri)
;  (princ "sum: ") (prin1 (stats0_sum ss)) (terpri)
  (princ "min: ") (prin1 (stats0_min ss)) (terpri)
  (princ "max: ") (prin1 (stats0_max ss)) (terpri)
  (princ "average: ") (prin1 (stats0_av ss)) (terpri)
  (princ "dev: ") (prin1 (stats0_dev ss)) (terpri)
  (let ((d (stats0_distr ss))
        (num (stats0_num ss)))
  (let ((cut1 (* .5 num))
        (cut (* .8 num))
        (tot 0))
    (dolist (x d)
      (let ((adjval (if (= 0 (car x)) (- (cdr x) zero) (cdr x))))
      (format t "~5d:  ~5d~%" (car x) adjval)
      (setq tot (+ tot adjval))
      (when (< cut tot) (princ "---------------------- 80%") (terpri)
        (setq cut num cut1 num))
      (when (< cut1 tot) (princ "---------------------- 50%") (terpri)
        (setq cut1 num))
      )
    )
  ))
  ))

(defun stats0_show_state (x unit)
  (let ((ss (stats0_glbl x)))
  (princ "number: ") (prin1 (stats0_num ss)) (terpri)
  (princ "min: ") (prin1 (stats0_min ss)) (terpri)
  (princ "max: ") (prin1 (stats0_max ss)) (terpri)
  (princ "average: ") (prin1 (stats0_av ss)) (terpri)
  (princ "dev: ") (prin1 (stats0_dev ss)) (terpri)
  (let ((d (stats0_distr ss)))
    (dolist (x d)
      (format t "~5d:  ~5d   " (car x) (cdr x))
      (let ((n (truncate (cdr x) unit)))
        (when (< 60 n) (princ "9") (setq n 60))
        (dotimes (i n) (princ "*"))
        (terpri))
    )
  )
  ))
```

```
(defvar whloc t)

(defun a_term_loc (x)
  (if whloc (si:address x)
  (let ((locs (get_an x 'where)))
    (if locs
      (car (car (last locs)))
      -1)
  )))

(defun find_empty_where (x)
  (if (or (term_is_var x) (term_is_const x)) nil
    (let ((res (null (get_an x 'where))))
    (when res
      (princ "--> ") (prin1 (a_term_loc x)) (princ ": ")
      (print_term_partial 2 x) (terpri))
    (dolist (e (term_args x))
      (setq res (or res (find_empty_where e)))
      )
    (when res
      (princ "<-- ") (prin1 (a_term_loc x)) (princ ": ")
      (print_term_partial 2 x) (terpri))
    res
    ))
  )

(defun find_it (x)
  (if (or (term_is_var x) (term_is_const x)) nil
    (progn
    (when (is_root x)
      (princ "--> ") (prin1 (a_term_loc x)) (princ ": ")
      (print_term_partial 2 x) (terpri))
    (dolist (e (term_args x))
      (find_it e)
      )
    ))
  )

(defun find_future (x)
  (if (or (term_is_var x) (term_is_const x)) nil
    (let ((res nil))
    (dolist (p (get_an x 'where) (setq res t))
      (when (<= (cdr p) clock) (return)))
    (when res
      (princ "--> ") (prin1 (a_term_loc x)) (princ ": ")
      (print_term_partial 2 x) (terpri))
    (dolist (e (term_args x))
      (setq res (or res (find_future e)))
      )
    (when res
      (princ "<-- ") (prin1 (a_term_loc x)) (princ ": ")
      (print_term_partial 2 x) (terpri))
    res
    ))
  )

(defun apply_rule (i r x)
  (set_an x 'tried t)
  (let ((mt (term_match (rule_lhs r) x)))
    (unless (eq 'fail mt)
    (if (rule_cond r)
        (let ((res (eval_cond_direct (rule_cond r) mt)))
        (if (eq 'fail res)
          (break "apply_rule: complex condition")
        (when res
          (setq rule_successful t)
          (ens_inc_repl current_ensemble)
          (incf replacements)
          (add_repl x r i (rule_inst r x mt)))))
      (progn
        (setq rule_successful t)
        (ens_inc_repl current_ensemble)
        (incf replacements)
        (add_repl x r i (rule_inst r x mt)))
    )
```

```
  )))

(defun find_untried (x)
  (if (or (term_is_var x) (term_is_const x)) nil
    (progn
    (unless (get_an x 'tried)
      (princ "--> ") (prin1 (a_term_loc x)) (princ ": ")
      (print_term_partial 2 x) (terpri))
    (dolist (e (term_args x))
      (find_untried e)
      )
    ))
  )

(defun find_tried (x)
  (if (or (term_is_var x) (term_is_const x)) nil
    (progn
    (when (get_an x 'tried)
      (princ "--> ") (prin1 (a_term_loc x)) (princ ": ")
      (print_term_partial 2 x) (terpri))
    (dolist (e (term_args x))
      (find_tried e)
      )
    ))
  )

(defun find_roots ()
  (dotimes (i rrm_size)
    (let ((inf (status_ensemble i)))
    (let ((rts (status_roots inf)))
    (when rts
      (princ "------------------------------------------------- ")
      (prin1 i) (terpri)
      (dolist (r rts)
        (prin1 (a_term_loc r)) (princ ": ") (print_term_partial 2 r)
        (terpri))
    ))))
  )

(defun show_not_tried (x)
  (if (all_tried x)
    (princ "#")
  (progn
  (when (< 60 (col)) (terpri) (princ "  "))
  (if (term_is_var x) (prin1 x)
  (if (term_is_const x) (prin1 (term_const_val x))
  (progn
    (princ "(")
    (prin1 (term_op x))
    (dolist (e (term_args x))
      (princ " ")
      (show_not_tried e))
    (when (< 60 (col)) (terpri) (princ "  "))
    (princ ")")
  ))))))

(defun all_tried (x)
  (if (or (term_is_var x) (term_is_const x))
    t
    (and (get_an x 'tried)
         (dolist (xa (term_args x) t)
           (unless (all_tried xa) (return nil))))
  ))

(defun find_in_roots (x)
  (dotimes (i rrm_size)
    (let ((inf (status_ensemble i)))
    (let ((rts (status_roots inf)))
    (when rts
      (dolist (r rts)
        (when (occ_in x r)
        (princ "ensemble: ") (prin1 i) (princ "  ")
        (prin1 (a_term_loc r)) (princ ": ") (print_term_partial 2 r)
        (terpri)))
    ))))
```

```
  )

(defun occ_in (x y)
  (if (eq x y) t
  (if (or (term_is_var y) (term_is_const y)) nil
  (if (is_root y) nil
    (dolist (e (term_args y))
      (when (occ_in x e) (return t)))
  ))))

(defun addr (x) (si:address x))

(defun selroot (e x)
  (nth x (status_roots (status_ensemble e)))
  )

(defun find_non_roots (x)
  (if (or (term_is_var x) (term_is_const x)) nil
    (progn
      (when (is_root x)
        (unless (find_a_root x)
          (prin1 (a_term_loc x)) (princ ": ") (print_term_partial 2 x)
          (terpri)
          ))
      (dolist (e (term_args x))
        (find_non_roots e))
      ))
  )

(defun show_a_root (x)
  (dotimes (i rrm_size)
    (let ((inf (status_ensemble i)))
    (let ((rts (status_roots inf)))
    (when rts
      (dolist (r rts)
        (when (eq x r)
          (princ " ") (prin1 i)))))))))

(defun find_a_root (x)
  (let ((res (find_a_root_1 x)))
    (prin1 (a_term_loc x)) (princ " --> ") (prin1 res) (terpri)
    res
    ))

(defun find_a_root_1 (x)
  (block whole
  (dotimes (i rrm_size)
    (let ((inf (status_ensemble i)))
    (let ((rts (status_roots inf)))
    (when rts
      (dolist (r rts)
        (when (eq x r) (return-from whole i)))))))))

(defvar check_list)

(defun not_tried ()
  (setq check_list (make-hash-table :size 5000 :test #'eq))
  (dotimes (e rrm_size)
    (let ((current_ensemble e))
    (let ((inf (status_ensemble e)))
    (dolist (r (status_roots inf))
      (traverse_tried r)
    )
  )))
  )

(defun traverse_tried (x)
; based on apply_rule_all
  (unless (or (term_is_var x) (term_is_const x))
    (setf (gethash x check_list) t)
    (dolist (e (term_args x))
      (unless (or (is_root e) ;@@ cut off rewriting at transition @@new
                  (not (occurs_in_ensemble e current_ensemble)))
      (traverse_tried e)
      ))
  )
  )

(defun ex () (exhibit_not_tried root t))

(defun exhibit_not_tried (x see)
  (if (every_tried x)
    (princ "$")
  (progn
  (when (< 60 (col)) (terpri) (princ "  "))
  (if (term_is_var x) (prin1 x)
  (if (term_is_const x) (prin1 (term_const_val x))
  (let ((flg (gethash x check_list)))
    (when (and see (not flg))
      (terpri)
      (find_in_roots x)
      (princ "<<")
      (prin1 (a_term_loc x))
      (princ ":")
      (print_term_partial 2 x)
      (princ ">>"))
    (princ (if flg "(" "["))
    (prin1 (term_op x))
    (dolist (e (term_args x))
      (princ " ")
      (exhibit_not_tried e flg))
    (when (< 60 (col)) (terpri) (princ "  "))
    (princ (if flg ")" "]"))
  ))))))

(defun every_tried (x)
  (if (or (term_is_var x) (term_is_const x))
    t
    (and (gethash x check_list)
         (dolist (xa (term_args x) t)
           (unless (every_tried xa) (return nil))))
  ))
```

```
; Numeric Fibo
(unless (fboundp 'simul) (load "ies"))

(to-file "tst4.out"

(setq tst 4)
(setq verbose 0)
(setq test-size 17)
(setq intra_cluster_penalty 2)
(setq inter_cluster_penalty 2)
(setq push_lim 3)
(setq build_lim 3)
(load "p")

(init)

(setq all_partitions '(
(
 (eq (fibo nil (0 nil)) (0 nil))
 (eq (fibo nil (1 nil)) (1 nil))
 (ceq (fibo nil n)
      (+ nil (fibo nil (- nil n (1 nil))) (fibo nil (- nil n (2 nil))))
      (and and (numberp nil n) (< nil (1 nil) n))
      )
 (eq (+ nil n (0)) n)
 (ceq (+ nil e1 e2) (b-i (mak (+ (car (val 'e1)) (car (val 'e2)))))
      (and nil (numberp nil e1) (numberp nil e2))
      )
 (ceq (- nil e1 e2) (b-i (mak (- (car (val 'e1)) (car (val 'e2)))))
      (and nil (numberp nil e1) (numberp nil e2))
      )
)
))

(let ((clock 0))
(setq root (init_term (rrm_ensemble 0 0 1 1) (list 'fibo (mak test-size))))
)

(princ "------------------------------------------------------------")
(princ " initial") (terpri)
(print_term root) (terpri)

(simul)

(load "a")
(v)
)
```

```
Loading p.lsp
rrm size: rrm clusters 4x4  cluster ensembles 4x4
stats: ens 10  clust ens 10  clust 10
intra cluster penalty: 2
inter cluster penalty: 2
gc period: 1
Loading alloc4.o
Finished loading alloc4.o
alloc threshold: 100
alloc randomization range: 100    cut: 2
when totally full: alloc in local cluster
Loading strat.o
Finished loading strat.o
intra cluster comm limit: 10
inter cluster comm limit: 40
size limit: 100
reloc size limit: 100
ensemble full threshold: 100
ensemble low level: 55
push lim: 20    build lim: 20
problem size: 30
clock limit: 999999
Finished loading p.lsp
------------------------------------------------------------ initial
(- 29 (- 28 (- 27 (- 26 (- 25 (- 24 (- 23 (- 22 (- 21 (- 20 (-
  19 (- 18 (- 17 (- 16 (- 15 (- 14 (- 13 (- 12 (- 11 (- 10 (-
  9 (- 8 (- 7 (- 6 (- 5 (- 4 (- 3 (- 2 (- 1 (- 0 (endl)))))))))
  ))))))))))))))))))))))))))))))
 0       40        0
 GC 1        0        0
 GC 2      140       40
 GC 3       40      100
 GC 4      100       40
 GC 5      140      140
 GC 6        0      100
 GC 7      140       40
 GC 8        0      100
 GC 9      140       40
 GC 10       0      100
 GC 11     140       40
 GC 12       0      100
 GC 13     140       40
 GC 14       0      100
 GC 15     220       40
 GC 16     120      100
 GC 17       0        0
 GC 18      80      120
 GC 19       0      120
 GC 20     240       80
 GC 21     120      200
 GC 22     100       40
 GC 23      80      100
 GC 24       0      120
 GC 25     200       80
 GC 26      40      200
 GC 27       0        0
 GC 28     140       40
 GC 29       0      100
 GC 30     140       40
 GC 31       0      100
 GC 32     220       40
 GC 33       0      100
 GC 34      80        0
 GC 35       0      120
 GC 36     240       80
 GC 37       0      200
 GC 38     140       40
 GC 39       0      100
 GC 40     140       40
 GC 41       0      100
 GC 42     140       40
 GC 43      40      100
 GC 44     140       40
 GC 45     140      140
 GC 46     140      140
 GC 47     140      140
 GC 48     140      140
 GC 49     220      140
 GC 50     140      140
 GC 51      80      100
 GC 52     140      160
 GC 53     200      180
 GC 54     180      240
 GC 55       0      100
 GC 56     280       80
 GC 57       0      200
 GC 58     280       80
 GC 59       0      200
 GC 60     360       80
 GC 61       0      200
 GC 62     220       40
 GC 63       0      220
 GC 64     340      120
 GC 65      40      300
 GC 66     140       40
 GC 67     140      140
 GC 68     140      140
 GC 69     140      140
 GC 70     100      140
 GC 71     220      140
 GC 72       0      100
 GC 73      40        0
 GC 74      40      120
 GC 75     100       40
 GC 76     140      140
 GC 77       0      100
 GC 78     140       40
 GC 79       0      100
 GC 80     220       40
 GC 81       0      100
 GC 82      80        0
 GC 83       0      120
 GC 84     200       80
 GC 85      40      200
 GC 86       0        0
 GC 87     140       40
 GC 88       0      100
 GC 89     140       40
 GC 90       0      100
 GC 91     220       40
 GC 92       0      100
 GC 93      80        0
 GC 94       0      120
 GC 95     240       80
 GC 96       0      200
 GC 97     140       40
 GC 98       0      100
 GC 99     140       40
 GC 100     40      100
 GC 101    140       40
 GC 102    140      140
 GC 103    140      140
 GC 104    140      140
 GC 105    140      140
 GC 106    220      140
 GC 107    100      140
 GC 108     80      100
 GC 109      0      120
 GC 110    240       80
 GC 111      0      200
 GC 112    140       40
 GC 113      0      100
 GC 114    140       40
 GC 115     40      100
 GC 116    100       40
 GC 117    140      140
 GC 118      0      100
 GC 119    140       40
 GC 120      0      100
 GC 121    220       40
```

```
GC 122          0        100
GC 123         80          0
GC 124          0        120
GC 125        240         80
GC 126          0        200
GC 127        180         40
GC 128          0        100
GC 129        240         80
GC 130          0        200
GC 131        140         40
GC 132          0        100
GC 133        140         40
GC 134          0        100
GC 135        140         40
GC 136          0        100
GC 137        140         40
GC 138          0        100
GC 139        100         40
Loading a.o
Finished loading a.o
number of state: 2304
--------------------------------------------------------------------------------
root: (- 0 (- 1 (- 2 (- 3 (- 4 (- 5 (- 6 (- 7 (- 8 (- 9 (- 10
  (- 11 (- 12 (- 13 (- 14 (- 15 (- 16 (- 17 (- 18 (- 19 (- 20
  (- 21 (- 22 (- 23 (- 24 (- 25 (- 26 (- 27 (- 28 (- 29 (endl
  ))))))))))))))))))))))))))))))                          .
--------------------------------------------------------------------------------
roots
      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 0:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 1:   .  1  2  .  .  .  .  .  .  .  .  .  .  .  .  .
 2:   .  o  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 3:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 4:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 5:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 6:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 7:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 8:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 9:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
10:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
11:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
12:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
13:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
14:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
15:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
--------------------------------------------------------------------------------
size
      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 0:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 1:   .  46 20 .  .  .  .  .  .  .  .  .  .  .  .  .
 2:   .  o  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 3:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 4:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 5:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 6:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 7:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 8:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 9:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
10:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
11:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
12:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
13:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
14:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
15:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
--------------------------------------------------------------------------------
maximum size
      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 0:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 1:   .  46 20 .  .  .  .  .  .  .  .  .  .  .  .  .
 2:   .  38 .  .  .  .  .  .  .  .  .  .  .  .  .  .
 3:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 4:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 5:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 6:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 7:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 8:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 9:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
10:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
11:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
12:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
13:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
14:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
15:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
--------------------------------------------------------------------------------
cluster matrix
      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 0:  ---  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1:   0 ---  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 2:   0  0 ---  0  0  0  0  0  0  0  0  0  0  0  0  0
 3:   0  0  0 ---  0  0  0  0  0  0  0  0  0  0  0  0
 4:   0  0  0  0 ---  0  0  0  0  0  0  0  0  0  0  0
 5:   0  0  0  0  0 ---  0  0  0  0  0  0  0  0  0  0
 6:   0  0  0  0  0  0 ---  0  0  0  0  0  0  0  0  0
 7:   0  0  0  0  0  0  0 ---  0  0  0  0  0  0  0  0
 8:   0  0  0  0  0  0  0  0 ---  0  0  0  0  0  0  0
 9:   0  0  0  0  0  0  0  0  0 ---  0  0  0  0  0  0
10:   0  0  0  0  0  0  0  0  0  0 ---  0  0  0  0  0
11:   0  0  0  0  0  0  0  0  0  0  0 ---  0  0  0  0
12:   0  0  0  0  0  0  0  0  0  0  0  0 ---  0  0  0
13:   0  0  0  0  0  0  0  0  0  0  0  0  0 ---  0  0
14:   0  0  0  0  0  0  0  0  0  0  0  0  0  0 ---  0
15:   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 ---
--------------------------------------------------------------------------------
alloc num: 13
alloc full: 0
alloc totally full: 0
--------------------------------------------------------------------------------
near communication conflicts: 0
far communication conflicts: 0
reloc size conflicts: 0
--------------------------------------------------------------------------------
ensemble full pushes: 0
ensemble full pushes with single root: 0
--------------------------------------------------------------------------------
replacement per step stats
number: 140
min: 0
max: 27
average: 3.2071428571428573
dev: 4.0470384737685983
    0:      6
    1:     52
    2:     40
---------------------- 50%
    3:     15
---------------------- 80%
    4:      7
    9:      3
   10:      6
   11:      2
   13:      4
   14:      4
   27:      1
--------------------------------------------------------------------------------
global ensemble stats
clock: 140
num: 256
used: 3
sw: 108
rel: 86
rel_far: 0
push: 11
repl: 449
max size: 46
num max size = 0: 253
av max size: 0.40625
av max size excluding 0: 34.666666666666664
size: 66
av size: 0.2578125
-------------------- av per tick
```

```
av used: 0.021428571428571429
av sw: 0.7714285714285714
av rel: 0.6142857142857143
av rel_far: 0.0
av push: 0.07857142857142857
av repl: 3.2071428571428573
-------------------- av per ensemble per tick
av used: 8.37053571428571 44E-5
av sw: 0.00301339285714285 73
av rel: 0.00239955357142857 16
av rel_far: 0.0
av push: 3.06919642857142 85E-4
av repl: 0.012527901785714286
--------------------------------------------------------------------------------
relocation within cluster
reloc cluster 0
          0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
    0:  ---
    1:      ---
    2:          ---
    3:              ---
    4:                  ---
    5:                      ---   1           1
    6:                        1 ---           2
    7:                                ---
    8:                                    ---
    9:                        1   1           ---
   10:                                            ---
   11:                                                ---
   12:                                                    ---
   13:                                                        ---
   14:                                                            ---
   15:                                                                ---
number of zero values: 234
global not zero number: 6
min: 0
max: 2
average: 1.1666666666666667
dev: 0.37267799624996456
     0:      0
     1:      5
---------------------- 80%
     2:      1

global number: 240
min: 0
max: 2
average: 0.029166666666666667
dev: 0.19144008346100236
     0:    234
---------------------- 80%
     1:      5
     2:      1

--------------------------------------------------------------------------------
pushes within cluster
push cluster 0
          0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
    0:  ---
    1:      ---
    2:          ---
    3:              ---
    4:                  ---
    5:                      ---   1           1
    6:                          ---
    7:                              ---
    8:                                  ---
    9:                                      ---
   10:                                          ---
   11:                                              ---
   12:                                                  ---
   13:                                                      ---
   14:                                                          ---
   15:                                                              ---
number of zero values: 238
global not zero number: 2
min: 0
max: 1
average: 1.0
dev: 0.0
     0:      0
     1:      2
---------------------- 80%

global number: 240
min: 0
max: 1
average: 0.008333333333333332
dev: 0.09090593428863 0946
     0:    238
---------------------- 80%
     1:      2

Loading a2.o
Finished loading a2.o
--------------------------------------------------------------------------------
global ensemble I/O state
input
num: 420
sum: 13360
min: 0
max: 220
av:  31.81
dev:  45.73
non zero av:  79.05
disribution:
     0:    251    59.76     0.00    59.76
    40:     74    17.62    43.79    77.38
    80:      1     0.24    44.38    77.62
   100:     71    16.90    86.39    94.52
   120:     10     2.38    92.31    96.90
   140:     10     2.38    98.22    99.29
   200:      2     0.48    99.41    99.76
   220:      1     0.24   100.00   100.00

output
num: 420
sum: 13360
min: 0
max: 220
av:  31.81
dev:  44.87
non zero av:  78.13
disribution:
     0:    249    59.29     0.00    59.29
    40:     74    17.62    43.27    76.90
    80:      2     0.48    44.44    77.38
   100:     75    17.86    88.30    95.24
   120:     10     2.38    94.15    97.62
   140:      8     1.90    98.83    99.52
   200:      1     0.24    99.42    99.76
   220:      1     0.24   100.00   100.00

--------------------------------------------------------------------------------
global per step I/O state
input
num: 140
sum: 13260
min: 0
max: 300
av:  94.71
dev:  57.55
non zero av: 102.00
disribution:
     0:     10     7.14     0.00     7.14
    40:     37    26.43    28.46    33.57
    80:     11     7.86    36.92    41.43
   100:     36    25.71    64.62    67.14
   120:     10     7.14    72.31    74.29
   140:     20    14.29    87.69    88.57
```

160:	1	0.71	88.46	89.29
180:	1	0.71	89.23	90.00
200:	11	7.86	97.69	97.86
220:	1	0.71	98.46	98.57
240:	1	0.71	99.23	99.29
300:	1	0.71	100.00	100.00

output
num: 140
sum: 13360
min: 0
max: 360
av: 95.43
dev: 87.12
non zero av: 145.22
disribution:

0:	48	34.29	0.00	34.29
40:	10	7.14	10.87	41.43
80:	8	5.71	19.57	47.14
100:	7	5.00	27.17	52.14
120:	2	1.43	29.35	53.57
140:	41	29.29	73.91	82.86
180:	2	1.43	76.09	84.29
200:	3	2.14	79.35	86.43
220:	9	6.43	89.13	92.86
240:	6	4.29	95.65	97.14
280:	2	1.43	97.83	98.57
340:	1	0.71	98.91	99.29
360:	1	0.71	100.00	100.00

tstl.out.pdl


```
Loading p.lsp
rrm size: rrm clusters 4x4  cluster ensembles 4x4
stats: ens 10  clust ens 10  clust 10
intra cluster penalty: 2
inter cluster penalty: 2
gc period: 1
Loading alloc4.lsp
Finished loading alloc4.lsp
alloc threshold: 100
alloc randomization range: 100    cut: 2
when totally full: alloc in local cluster
Loading strat.lsp
Finished loading strat.lsp
intra cluster comm limit: 10
inter cluster comm limit: 40
size limit: 100
reloc size limit: 100
ensemble full threshold: 100
ensemble low level: 55
push lim: 20    build lim: 20
problem size: 7
clock limit: 999999
Finished loading p.lsp
t

>nil

>(setq all_partitions '(
(
  (ceq (- nil e1 (- nil e2 1)) (- nil e2 (- nil e1 1)) (< nil e2 e1))
)
))

(let ((clock 0) (term '(endl)))
 (dotimes (i test-size) (setq term (list '- (list i) term)))
(setq root (init_term (rrm_ensemble 0 0 1 1) term))
)
(((ceq (- nil e1 (- nil e2 1)) (- nil e2 (- nil e1 1)) (< nil e2 e1))))

>(- ((where (17 . 0))) (6 ((where (17 . 0))))
   (- ((where (17 . 0))) (5 ((where (17 . 0))))
      (- ((where (17 . 0))) (4 ((where (17 . 0))))
         (- ((where (17 . 0))) (3 ((where (17 . 0))))
            (- ((where (17 . 0))) (2 ((where (17 . 0))))
               (- ((where (17 . 0))) (1 ((where (17 . 0))))
                  (- ((where (17 . 0))) (0 ((where (17 . 0))))
                     (endl ((where (17 . 0))))))))))))

>(princ "------------------------------------------------------------")
(princ " initial") (terpri)
(print_term root) (terpri)

------------------------------------------------------------
"------------------------------------------------------------"

> initial
" initial"

>
nil

>(- 6 (- 5 (- 4 (- 3 (- 2 (- 1 (- 0 (endl)))))))
")"

>
nil

>
(simul)


 0 GC 1 GC 2 GC 3 GC 4 GC 5 GC 6 GC 7
nil

>(load "a")

Loading a.lsp
Finished loading a.lsp
t

>(v)
number of states: 1792
------------------------------------------------------------------------
root: (- 0 (- 1 (- 2 (- 3 (- 4 (- 5 (- 6 (endl))))))))
------------------------------------------------------------------------
roots
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 0:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 1:|  .  1  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 2:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 3:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 4:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 5:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 6:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 7:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 8:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 9:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
10:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
11:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
12:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
13:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
14:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
15:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
------------------------------------------------------------------------
size
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 0:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 1:|  . 15  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 2:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 3:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 4:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 5:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 6:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 7:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 8:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 9:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
10:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
11:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
12:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
13:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
14:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
15:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
------------------------------------------------------------------------
maximum size
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 0:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 1:|  . 15  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 2:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 3:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 4:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 5:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 6:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 7:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 8:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
 9:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
10:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
11:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
12:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
13:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
14:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
15:|  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
------------------------------------------------------------------------
cluster matrix
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 0:| --- 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0|
 1:|  0 --- 0  0  0  0  0  0  0  0  0  0  0  0  0  0|
 2:|  0  0 --- 0  0  0  0  0  0  0  0  0  0  0  0  0|
 3:|  0  0  0 --- 0  0  0  0  0  0  0  0  0  0  0  0|
 4:|  0  0  0  0 --- 0  0  0  0  0  0  0  0  0  0  0|
 5:|  0  0  0  0  0 --- 0  0  0  0  0  0  0  0  0  0|
```

```
 6: |   0   0   0   0   0   0 ---   0   0   0   0   0   0   0   0   0 |
 7: |   0   0   0   0   0   0   0 ---   0   0   0   0   0   0   0   0 |
 8: |   0   0   0   0   0   0   0   0 ---   0   0   0   0   0   0   0 |
 9: |   0   0   0   0   0   0   0   0   0 ---   0   0   0   0   0   0 |
10: |   0   0   0   0   0   0   0   0   0   0 ---   0   0   0   0   0 |
11: |   0   0   0   0   0   0   0   0   0   0   0 ---   0   0   0   0 |
12: |   0   0   0   0   0   0   0   0   0   0   0   0 ---   0   0   0 |
13: |   0   0   0   0   0   0   0   0   0   0   0   0   0 ---   0   0 |
14: |   0   0   0   0   0   0   0   0   0   0   0   0   0   0 ---   0 |
15: |   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 --- |
--------------------------------------------------------------------------------
alloc num: 0
alloc full: 0
alloc totally full: 0
--------------------------------------------------------------------------------
near communication conflicts: 0
far communication conflicts: 0
reloc size conflicts: 0
--------------------------------------------------------------------------------
ensemble full pushes: 0
ensemble full pushes with single root: 0
--------------------------------------------------------------------------------
replacement per step stats
number: 8
min: 0
max: 6
average: 3.0
dev: 1.5
    0:      1
    3:      6
---------------------- 80%
    6:      1
--------------------------------------------------------------------------------
global ensemble stats
clock: 8
num: 256
used: 1
sw: 1
rel: 0
rel_far: 0
push: 0
repl: 24
max size: 15
num max size = 0: 255
av max size: 0.05859375
av max size excluding 0: 15.0
size: 15
av size: 0.05859375
-------------------- av per tick
av used: 0.125
av sw: 0.125
av rel: 0.0
av rel_far: 0.0
av push: 0.0
av repl: 3.0
-------------------- av per ensemble per tick
av used: 4.8828125E-4
av sw: 4.8828125E-4
av rel: 0.0
av rel_far: 0.0
av push: 0.0
av repl: 0.01171875
--------------------------------------------------------------------------------
relocation within cluster
--------------------------------------------------------------------------------
pushes within cluster
Loading a2.lsp
Finished loading a2.lsp
--------------------------------------------------------------------------------
global ensemble I/O stats
input
num: 8
sum: 0
min: 0
max: 0
av:   0.00
dev:   0.00
disribution:
    0:      8   100.00     0.00   100.00

output
num: 8
sum: 0
min: 0
max: 0
av:   0.00
dev:   0.00
disribution:
    0:      8   100.00     0.00   100.00

--------------------------------------------------------------------------------
global per step I/O stats
input
num: 8
sum: 0
min: 0
max: 0
av:   0.00
dev:   0.00
disribution:
    0:      8   100.00     0.00   100.00

output
num: 8
sum: 0
min: 0
max: 0
av:   0.00
dev:   0.00
disribution:
    0:      8   100.00     0.00   100.00

nil

>
```

B Detailed Description of the New Ensemble Simulator

The New RISC Ensemble Simulator

Tim Winkler

Abstract: The purpose of these notes is to document the new RISC ensemble simulator.

1 Introduction

The outline of the rest of this report is this:

- Overview of simulator.
- Basic structures.
- Basic simulation process: a simulation step.
- Instruction set summary.
- Annotated code for operations.

This discussion only includes the basic simulator.

2 Overview

The ensemble simulator is written in C and as a whole consists of

- `risc.h` — which contains the basic parameters, macro definitions, type definitions, and global variable declarations.
- `risc.c` — This contains the `main` routine, `simd_execute` called from the main routine (which executes the simd code) and is defined in the user provided SIMD code, and the basic SIMD execution routines (`simd`, `simd1`, ...). Supporting the SIMD execution are various routines in `basic.c` which are not discussed here. These are all part of the software of the ensemble and do not affect the design of the ensemble. Normally the user of the simulator will put data graph initialization and the routine `simd_execute` in the file `simd.c` and link this with the rest of the simulator.
- `basic.h`, `basic.c` less basic routines (in fact, these can be viewed as providing SIMD routines) that are not discussed here.

3 Basic Structures

The basic structures used in the simulation are variables that represent registers and wires/buses. The value of a variable representing a bus indicates the currently asserted state of the bus. Normally the value asserted

should not persist beyond one whole clock cycle, and should only change once during a clock phase (roughly speaking).

Here are some basic constants. From `risc.h`:

```
/* ensemble configuration */

#define COLNUM 12  — Number of columns of tiles.
#define ROWNUM 12  — Number of rows of tiles.
/* Are row and col numbers powers of two corresponding to ADDRSHIFTs? */
#define POW2 0  — See comment above.
#define TILENUM (COLNUM * ROWNUM)  — Total number of tiles.
#define CELLSPERTILE 4  — Number of cells per tile.
#define CELLNUM (TILENUM * CELLSPERTILE)  — Total number of cells.
#define PORTNUM (2*TILENUM + COLNUM + ROWNUM)  — Number of ports.

#define PORTCNT 4  — Number of ports at cell.
```

Here are the central type defintions. Note: the ids are not used in the simulation process and were included to make structures self-identifying.

```
/* type definitions */

/* only used to get at id (really) */
typedef struct genericstruct {  — Any structure below looks like this.
  int genid;
  int genval; /* only because is a common case */
} genericthing;

typedef struct portstruct {  — Represents a port bus.
  int portid;
  int portaddr;
  int portval;
  int portloc;
} portbus;

typedef struct colstruct {  — Represents a column bus.
  int colid;
  int colval;
  int colowner; /* need not exist in actual hardware */
} colbus;
#define FAILFLAG (1<<30)

typedef struct rowstruct {  — Represents a row bus.
  int rowid;
  int rowval;
  int rowowner;  /* need not exist in actual hardware */
} rowbus;

typedef struct cellstruct {  — Represents a cell.
  int self; /* own address */
  int reg[REGCNT];  — Includes ACC.
  int marks;
  portbus *ports[PORTCNT];
```

```
  colbus *col;
  rowbus *row;
  int portno;  — Value 0..7; unique for each cell on port bus.
  int refc;  — Reference count.
  int aux;  — Used in SELCELL MAKEREQ FETCHDATA, etc.
  int state;  — The state flags (see below).
} cellstate;
/* some defines related to the above */
#define CELLTOK reg[TOK]
#define CELLFLAGS reg[FLAGS]
#define CELLLEFT reg[LEFT]
#define CELLRIGHT reg[RIGHT]
#define CELLNTOK reg[NTOK]
#define CELLNFLAGS reg[NFLAGS]
#define CELLNLEFT reg[NLEFT]
#define CELLNRIGHT reg[NRIGHT]
#define CELLACC reg[ACC]

#define PORTE ports[EAST]
#define PORTN ports[NORTH]
#define PORTW ports[WEST]
#define PORTS ports[SOUTH]

#define STATE_VALID 1
/*  VALID means allocated and allowed to become active */
#define STATE_ACTIVE (1<<1)
#define STATE_TRYING (1<<2)
#define STATE_REMOTE_LEFT (1<<5)
#define STATE_REMOTE_RIGHT (1<<6)
#define STATE_COMMIT (1<<7)

typedef struct simdstruct {  — SIMD broadcast bus.
  int id;
  int op;  — Current operation/opcode.
  int arg1,arg2,arg3,arg4;  — arguments to operation.
  int ph;  — Clock phase, 1 or 2.
} simdbus;
```

There are 32 marks in a cell, referred to by index (0–31). There may be some additional state bits: WORKING, ZERO, MINUS, CARRY, OVERFLOW, and FAILURE. The aux register might be the ACC.

Here are the actual declarations of the key shared variables.

```
extern int clock; /* Note: not phase number */

extern int globalfeedback;

extern simdbus simdinstr;

extern portbus port[PORTNUM];

extern colbus col[COLNUM];
extern rowbus row[ROWNUM];

extern cellstate cell[CELLNUM];
```

```
int randomstate;
```

4 The Simulation Process

The user code calls the **simd**, **simd1**, ... routines, and also uses the **resetglob** and **testglob** routines; these routines give control the the ensemble. The user SIMD routines may also use whatever controller state is desired (e.g. flags).

Execution of

```
simd2(TSTMARKPTR,1,LEFT); SHOW;
```

causes the opcode (**TSTMARKPTR**) and arguments (**1**,**LEFT**) to be put into the **simdinstr** structure that represents the SIMD control broadcast bus, and then the execution of **simdstep**. There are many extranenous details to **simdstep** (e.g., recording frequency of execution of instructions), but the key process is to

- Increment the clock.
- Reset port, row, and col buses (if needed).
- Set the instruction phase to 1 (**simdinstr.ph = 1**).
- Perform **simdaction** for every cell.
- In the case of **CELLARBPTR** and **ARBROW**, perform some inter-clock phase actions that cannot easily be associated with cell actions.
- Set the instruction phase to 1 (**simdinstr.ph = 2**).
- Perform **simdaction** for every cell.

The **simdaction** routine takes a pointer to cell and uses a large switch statement to execute the code associated with the SIMD op.

5 Basic definitions

The following are very basic bit operations:

```
/* general defines */
  — The are bit testing operations based on bit indices.
#define TSTBITN(x,n) ((x)&(1<<(n)))
#define ADDBITN(x,n) ((x)|(1<<(n)))
#define REMBITN(x,n) ((x)& ~(1<<(n)))
#define ASNBITN(x,n,y) ((y)?ADDBITN(x,n):REMBITN(x,n))
#define MSKBITN(n) (1<<(n))
#define SETBITN(x,n) ((x) |= (1<<(n)))
#define CLRBITN(x,n) ((x) &= ~(1<<(n)))

#define LOWERBITS(n) ((1<<(n))-1)
```

— *The are bit testing operations based on bit masks.*

```
#define TSTBIT(x,n) ((x)&(n))
#define ADDBIT(x,n) ((x)|(n))
#define REMBIT(x,n) ((x)&(~(n)))
#define ASNBIT(x,n,y) ((y)?ADDBIT(x,n):REMBIT(x,n))
#define SETBIT(x,n) ((x) |= (n))
#define CLRBIT(x,n) ((x) &= ~(n))
/* Note: duplication of n in following: */
#define TSTALLBIT(x,n) ((n) == ((x)&(n)))
#define LOWESTBIT(n) ((n)&-(n))
#define ATMOSTONEBIT(n) ((n)&~-(n))

#define UNDEF (0xbadbadee)
```

The following are the basic definitions related to addresses, address mappings, port selection, and PORTNO calculations.

```
/* basic defines */

/* Addresses: top left is 0,0 (is X,Y); X is row and Y is col */

/*      Y=0  1  2  3  4  5  6  7   */
/*   X=0  0  1  2  3  4  5  6  7   */
/*     1  8  9 10 11 12 13 14 15   */
/*     2 16 17 18 ....             */

#define ADDRSHIFT 4  — Bits to represent column or row.
#define ADDRTOPSHIFT 8
/*  should be 2*ADDRSHIFT */

/* Changed cell nums by +1 in addrs */
  — Compute address from row,col, cell num.
#define ADDR(x,y,n) ((((n)+1) << ADDRTOPSHIFT) + ((x)<<ADDRSHIFT)+(y))
  — Tile is just row,col, no cell number.
#define TILE(x) ((x) & ((1<<ADDRTOPSHIFT)-1))
  — Decode addresses.
#define ROW(x) (((x)>>ADDRSHIFT) & ((1<<ADDRSHIFT)-1))
#define COL(x) ((x) & ((1<<ADDRSHIFT)-1))
#define NUM(x) (((x) >> ADDRTOPSHIFT)-1)
#define CELLNUMINCR (1<<ADDRTOPSHIFT)
#define TILEADDR(x,y) (((x)<<ADDRSHIFT)+(y))

  — Translating addresses to indices.
#if POW2

#define CELLIND(x,y,n) (((n) << ADDRTOPSHIFT) + ((x)<<ADDRSHIFT)+(y))
/* convert address to cell[] index */
#define ADDRTOIND(x) ((x)-CELLNUMINCR)
#define TILEIND(x,y) (((x)<<ADDRSHIFT)+(y))

#else

#define CELLIND(x,y,n) (((n)*ROWNUM +  (x))*COLNUM + (y))
/* convert address to cell[] index */
#define ADDRTOIND(x) ((NUM(x)*ROWNUM +  ROW(x))*COLNUM + COL(x))
```

```
#define TILEIND(x,y) (((x)*COLNUM)+(y))

#endif

#define ROWINCR (1<<ADDRSHIFT)
#define COLINCR 1

#define ADDRUP(x) ((x)-ROWINCR)  — Adjacent tiles.
#define ADDRDWN(x) ((x)+ROWINCR)
#define ADDRLFT(x) ((x)-COLINCR)
#define ADDRRGT(x) ((x)+COLINCR)

#define CELLAT(a) cell[ADDRTOIND(a)]  — Address to cell.
#define CELLREF(x,y,n) cell[CELLIND(x,y,n)]  — r,c,n to cell.

  — Indices for ports.
#define PORTNIND(x,y) (2*TILEIND(x,y))
#define PORTWIND(x,y) (2*TILEIND(x,y)+1)
#define ISPORTW(x) ((x)&1)
/*   only when not on right or bottom edge */
#define PORTEIND(x) (2*TILENUM+(x))
/*   along right side */
#define PORTSIND(y) (2*TILENUM+ROWNUM+(y))
/*   along bottom */

/* access to all ports by allowing "the next index over" */
#define PORTNINDX(x,y) ((x)==ROWNUM ? PORTSIND(y) : PORTNIND(x,y))
#define PORTWINDX(x,y) ((y)==COLNUM ? PORTEIND(x) : PORTWIND(x,y))

/* not really correct: */
#if POW2
#define VALIDADDR(x) (CELLNUMINCR<=(x) && \
                      (x)<(CELLNUM+CELLNUMINCR))
#else
#define VALIDADDR(x) (ADDR(0,0,0)<=(x) && \
    (x)<ADDR(ROWNUM-1,COLNUM-1,CELLSPERTILE-1) && \
    (COL(x)<COLNUM) && (ROW(x)<ROWNUM))
#endif

#define VALIDDIR(x) (0<=(x) && (x)<= 3)

/* a pair of tileaddrs */
/* Try to treat addr as local when same tile? */
#define ISLOCAL(x,y) (((x)<=(y))? ((COLINCR==((y)-(x)) && COL(y)!=0) \
                                   || ROWINCR==((y)-(x))) : \
    ((COLINCR==((x)-(y)) && COL(x)!=0) || ROWINCR==((x)-(y))))

  — Compute direction of y from x (or undef).
#define WHICHDIR(x,y) \
   ((COLINCR==((y)-(x)))?((COL(y)==0)? UNDEF : EAST): \
    (ROWINCR==((y)-(x)))?SOUTH: \
    (-COLINCR==((y)-(x)))?((COL(x)==0)? UNDEF : WEST): \
    (-ROWINCR==((y)-(x)))?NORTH: UNDEF)
```

```
/* addr -> portno */
#define PORTNO(a) ((1&((a)+((a)>>ADDRSHIFT)))?NUM(a)+4:NUM(a))
#define PORTNOSHIFT 8
#define PORTNOCNT 8
#define PORTNOCASE(a) ((a)<CELLSPERTILE)
#define PORTNOSELBIG(a,b) (((a)<CELLSPERTILE) ? ((b)>>PORTNOSHIFT) : \
    ((b)&LOWERBITS(PORTNOSHIFT)))
#define PORTNOSEL(a,b) ((a) ? ((b)>>CELLSPERTILE) : \
    ((b)&LOWERBITS(CELLSPERTILE)))
#define PORTNOREPLY(a,b) (((a)<CELLSPERTILE) ? ((b)<<CELLSPERTILE) : (b))

/* ROW and COL work for PORTADDRs too */
#define PORTADDR(x,y,d) (((d) << ADDRTOPSHIFT) + ((x)<<ADDRSHIFT)+(y))
#define PORTDIR(x) ((x) >> ADDRTOPSHIFT)

/* number of registers in one set */
#define REGNUM 4
```

These are some simple defines used to make the SIMD code a bit more readable.

```
/* registers */
#define TOK 0
#define FLAGS 1
#define LEFT 2
#define RIGHT 3
#define NTOK 4
#define NFLAGS 5
#define NLEFT 6
#define NRIGHT 7
#define ACC 8
#define REGCNT 9

/* directions */
#define EAST 0
#define NORTH 1
#define WEST 2
#define SOUTH 3
```

In fact, it is possible to use 16 registers by changing `REGCNT` to 16. One may also want to adjust `REGNUM`.

6 Instruction Set Summary

Specifiers for arguments to SIMD instructions:

- `val`: 16 (actually 32 bit) value
- `mark`: 0–31 bit position
- `reg`, `ptr`, `source-ptr`, `target-ptr`, `sreg`, `treg`, `source`, `target`: 0–8 register selector
  May be 0–15 instead.
- `dir`: 0=EAST, 1=NORTH, 2=WEST, 3=SOUTH
- `portno`: 0–7 select active element on port

Basic instructions

NOP — *Do nothing.*

INIT — *Activate all allocated.*
ERASE — *Clear marks.*

CONST val — *val to* ACC.
CLEAR reg — *0 to reg.*

EQ reg reg — *Various tests.*
NEQ reg reg
GT reg reg
LE reg reg
LT reg reg
GE reg reg
TSTZERO reg
TSTNZERO reg

MOVE treg sreg — *From source to target.*
ADD treg sreg
SUB treg sreg
LOGNOT reg
LOGAND treg sreg
LOGIOR treg sreg
LOGXOR treg sreg

(For these next, probably want to use a state bit for bits shifted out)
ROTATEL treg sreg — *16-bit rotate left.*
ROTATER treg sreg — *16-bit rotate right*
SHIFTL treg sreg — *Shift left..*
SHIFTR treg sreg — *Shift right.*
SHIFTRA treg sreg — *Shift right arithmetic (preserve sign).*

SETRANDOM treg — *Put in 16 (32) bit psuedo-random value.*

TSTMARK mark — *Local mark operations.*
TSTNOTMARK mark
SETMARK mark
CLRMARK mark

TSTMARKPTR mark source-ptr — *Adjacent tile mark operations.*
TSTNOTMARKPTR mark source-ptr
SETMARKPTR mark target-ptr
CLRMARKPTR mark target-ptr

INCR val — *Increment reference count of active cell.*
val can be +1,-1
INCRPTR1 val target-ptr — *RISC incrptr part 1 (cells 0,1).*
INCRPTR2 val target-ptr — *RISC incrptr part 2 (cells 2,3).*

SETTRYING — *Set state trying flag.*
Performs global feedback on any active.
INITTRYING — *Equiv. to:* INIT, *test* TRYING *flag.*

Performs global feedback on any active.
`CLRTRYING` — *Clear state trying flag.*

`ARBPORTPTR reg` — *Arbitrate for port based on pointer, clear TRYING if succeed (deactivate if non-local).*
`ARBPORT dir` — *Arbitrated on given port clear* `TRYING` *if succeed.*

`MAKEREQ ptr` — *RISC fetch part 1; uses aux.*
May want a version, `MAKEREQDIR` *dir, that goes in a certain direction (not defined yet).*
`FETCHDATA treg source-ptr sreg` — *RISC fetch part 2.*

`STORE target-ptr treg sreg` — *RISC store (assumes arbitration).*
[SHOULD BE SPLIT INTO MAKEREQ and STOREDATA.]

`STOREPN portno target-ptr treg sreg` — *Portno activation, no arb. needed.*
`FETCHPN portno treg source-ptr sreg` — *Portno activation, no arb. needed.*

`CELLARBPTR target-ptr` — *After* `ARBPORTPTR`*, do cell arbitration (not used).*

`ALLOCREQ` — *RISC alloc part 1; uses aux.*
`AVAIL1 reg` — *RISC alloc part 2 (cells 0,1).*
`AVAIL2 reg` — *RISC alloc part 3 (cells 2,3).*
`ALLOCPTR target-ptr` — *RISC alloc part 4; allocate pointed-at cell.*

`SENDGLOBAL` — *Initiate global feedback.*
`resetglob()` — *CONTROLLER CODE: reset global feedback indicator.*
`testgloba()` — *CONTROLLER CODE: test global feedback indicator.*

ROW/COL operations

`ARBTILE` — *ROW/COL part 1; equiv. to* `ARBPORT NORTH`*, but doesn't affect* `TRYING`.
`ARBCOL` — *ROW/COL part 2.*
`SELROW target` — *ROW/COL part 3.*
`ARBROW target` — *ROW/COL part 4.*
`SELCELL ptr` — *ROW/COL part 5.*

`FETCHGLBL target source-ptr reg` — *Global actiions.*
`STOREGLBL target-ptr reg source`
`TSTMARKGLBL source-ptr mark`
`TSTNOTMARKGLBL mark source-ptr`
`SETMARKGLBL mark target-ptr`
`CLRMARKGLBL mark target-ptr`
`INCRGLBL val target-ptr`

Other instructions

`SWAP` — *Switch* `TOK`, `NTOK`, *etc.*
`COMMIT` — `SWAP`*; set* `COMMIT` *flag.*
`COMMITMARK mark` — *If mark, then* `SWAP`*; always set* `COMMIT` *flag.*
`SETCOMMIT` — *Set* `COMMIT` *flag.*

`TSTSTATE flags` — *Operations on state flags (use masks not bit positions).*
`TSTNOTSTATE flags`

SETSTATE flags
CLRSTATE flags

TSTLOCAL ptr — *Test if local pointer.*
TSTREMOTE ptr — *Test if remote pointer.*
TSTADDR ptr — *Test if appears to be an address.*
TSTNOTADDR ptr — *Test if appears not to be an address.*

TSTINUSE — *Test is reference count is positive (¿=1).*
TSTUNUSED — *Test if reference count is not positive (¡=0).*
TSTUNSHARED — *Test if reference count is 1.*

RELOCREQ ptr — *If local, succeed, otherwise start relocation.*
(Set REMOTE_ flags.)

DELETE — *Clear registers, marks, and state.*

INITALL — *Init all including unallocated.*
TSTBLACK — *Checkerboard activation.*
TSTRED — *Checkerboard activation.*
TSTCLASS class -- One of 5 cases activation, class: 0-4.
TSTCELLNUM cellnum — *activate based on cell number: 0-3.*

7 Annotated Code

The following is the key simulator code with some annotations.

This is the initialization routine. The ids of objects are filled in (although these are not currently used). The central function is to link up the cells with the port buses.

```
ensemble_init()
{
  register int i,j,n,a;
  cellstate *cp;

  simdinstr.id = 0;
  simdinstr.op = NOP;
  simdinstr.arg1 = UNDEF;
  simdinstr.arg2 = UNDEF;
  simdinstr.arg3 = UNDEF;
  simdinstr.arg4 = UNDEF;

  for (i=0; i<PORTNUM; i++) {
    port[i].portid = MKID(PORT,i);
    port[i].portloc = 0;
  }
  for (i=0; i<ROWNUM; i++) row[i].rowid = MKID(ROWS,i);
  for (i=0; i<COLNUM; i++) col[i].colid = MKID(COLS,i);

  for (i=0; i<ROWNUM; i++)
    for (j=0; j<COLNUM; j++) {
      for (n=0; n<CELLSPERTILE; n++) {
```

```
        a = ADDR(i,j,n);
        cp = &CELLAT(a);
        cp->self = MKID(CELL,a);
        /* used when all cells on a port need to use a bit, etc. */
        cp->portno = n + ((1&(i+j)) ? CELLSPERTILE : 0);
        cp->row = &row[i];  — Row and col are easy.
        cp->col = &col[j];
           — Ports are more complex.
        cp->PORTN = &port[PORTNINDX(i,j)];
        cp->PORTN->portloc = PORTADDR(i,j,NORTH);
        cp->PORTW = &port[PORTWINDX(i,j)];
        cp->PORTW->portloc = PORTADDR(i,j,WEST);
        cp->PORTE = &port[PORTWINDX(i,j+1)];
        if (j+1 == COLNUM) cp->PORTE->portloc = PORTADDR(i,j,EAST);
        cp->PORTS = &port[PORTNINDX(i+1,j)];
        if (i+1 == ROWNUM) cp->PORTS->portloc = PORTADDR(i,j,SOUTH);
        cp->marks = 0;
        cp->refc = 0;
        cp->state = 0;
        /* { int m; for (m=0; m<REGCNT; m++) cp->reg[m] = 0; } */
      }
    }
}
```

Here are the basic SIMD broadcast routines. Simply copy arguments to **simdinstr** structure. (Would have liked a somewhat different approach to variable numbers of arguments than provided in C.)

```
simd(op)
int op;
{
  simdinstr.op = op;
  simdinstr.arg1 = UNDEF;
  simdinstr.arg2 = UNDEF;
  simdinstr.arg3 = UNDEF;
  simdinstr.arg4 = UNDEF;
  simdstep();
}
simd1(op,arg)
int op,arg;
{
  simdinstr.op = op;
  simdinstr.arg1 = arg;
  simdinstr.arg2 = UNDEF;
  simdinstr.arg3 = UNDEF;
  simdinstr.arg4 = UNDEF;
  simdstep();
}
simd2(op,arg1,arg2)
int op,arg1,arg2;
{
  simdinstr.op = op;
  simdinstr.arg1 = arg1;
  simdinstr.arg2 = arg2;
  simdinstr.arg3 = UNDEF;
```

```
  simdinstr.arg4 = UNDEF;
  simdstep();
}
simd3(op,arg1,arg2,arg3)
int op,arg1,arg2,arg3;
{
  simdinstr.op = op;
  simdinstr.arg1 = arg1;
  simdinstr.arg2 = arg2;
  simdinstr.arg3 = arg3;
  simdinstr.arg4 = UNDEF;
  simdstep();
}
simd4(op,arg1,arg2,arg3,arg4)
int op,arg1,arg2,arg3,arg4;
{
  simdinstr.op = op;
  simdinstr.arg1 = arg1;
  simdinstr.arg2 = arg2;
  simdinstr.arg3 = arg3;
  simdinstr.arg4 = arg4;
  simdstep();
}
```

This is the core SIMD execution routine (the process of execution has already been briefly discussed). The important state changes involve `clock`, `simdinst.ph`, and the commnication buses.

```
simdstep()
{
  register int i,op,p,flg,val;

  clock++;  — Update clock.

  op = simdinstr.op;  — Get op.
  if (recordinstrfreq) {
    if (op != INIT) instrfreq[op]++;
  }
  if (showinstrs) {
    printf("%d: ",clock);
    printop(); printf("\n"); fflush(stdout); }

  if (op != NOP) {
  /* Actually there seem to be cases here */
  /* arbcol, arbrow are actions over different entities */

  /* Could improve on this */
  if (op==ARBCOL)  — Conditionally reset row/col buses.
    for (i=0; i<COLNUM; i++) { col[i].colval = 0; col[i].colowner = 0; }
  else
  if (op==SELROW)
    for (i=0; i<COLNUM; i++) col[i].colval = UNDEF;
  else
  if (op==SELCELL)
    for (i=0; i<CELLNUM; i++) cell[i].aux = 0;
```

```
   — Always reset port buses.
for (i=0; i<PORTNUM; i++) port[i].portval = 0;
for (i=0; i<PORTNUM; i++) port[i].portaddr = UNDEF;

/* if (verbose) printf("phase1\n"); */

simdinstr.ph = 1;  — Perform phase 1, all cells.
for (i=0; i<CELLNUM; i++)
  simdaction(&cell[i]);

/* if (verbose) printf("port\n"); */
/* In the following could put checks to see if bus already non-UNDEF
   which would indicate an error in use; this also ignores issues
   of cell arb */
if (op == FETCH) {  — Not currently used.
  for (i=0; i<PORTNUM; i++)
    if (port[i].portaddr != UNDEF)
      port[i].portval = CELLAT(port[i].portaddr).reg[simdinstr.arg3];
}
else
if (op == CELLARBPTR) {
  for (i=0; i<CELLNUM; i++) {
    flg = 0;
    val = MSKBITN(cell[i].portno);
    for (p=0; p<PORTCNT; p++) {
      if (TSTBIT(cell[i].ports[p]->portval,val)) {
        if (flg) CLRBIT(cell[i].ports[p]->portval,val);
        flg = 1;
      }
    }
  }
}
else
if (op == ARBROW) {
  for (i=0; i<ROWNUM; i++) { row[i].rowval = UNDEF; row[i].rowowner = 0; }
  for (p=COLNUM-1; 0<=p; p--) {
    if (0 != col[p].colowner) {
      val = col[p].colval;
      if (val != UNDEF) {
        i = row[val].rowowner;
        if (i != 0) SETBIT(col[COL(i)].colval,FAILFLAG);
        row[val].rowowner = col[p].colowner;
      }
    }
  }
}


/* if (verbose) printf("phase2\n"); */
simdinstr.ph = 2;  — Perform phase 2, all cells.
for (i=0; i<CELLNUM; i++)
  simdaction(&cell[i]);
}
```

```
}
```

This routine, **simdaction**, is the heart of it all. It gets a pointer to a cell and uses the clock phase and the SIMD broadcast to perform the appropriate action on the cell and busses. Here is the prolog. The large **switch** statement is discussed below. Note the "local"' macros that are introduced.

```
simdaction(cp)
register cellstate *cp;
{
  register int op,ph,st,r,val;
  int dir,tile,mask,a,flg;

  /* Do some top-level case analysis depending on whether active, etc. */
  st = cp->state;
#define DEACTIVATE CLRBIT(cp->state,STATE_ACTIVE)
#define ACTIVATE SETBIT(cp->state,STATE_ACTIVE)
  op = simdinstr.op;
  ph = simdinstr.ph;
#define IS_ACTIVE TSTBIT(st,STATE_ACTIVE)
#define IS_VALID TSTBIT(st,STATE_VALID)

  switch (op) {
    — All the case are discussed below.
```

Some cases in the **switch** are omitted. The various operations are discussed below.

Here is a large group of relatively trivial operations that are not discussed in detail. These instructions consist of simple local operations and tests.

```
  case NOP: break;
  case INIT:
    if (ph==2 && TSTBIT(st,STATE_VALID)) ACTIVATE;
    break;
  case ERASE:
    if (ph==2 && IS_ACTIVE) cp->marks = 0;
    break;
  case TSTMARK: /* mark */
    if (ph==2 && IS_ACTIVE && !TSTBITN(cp->marks,simdinstr.arg1))
      DEACTIVATE;
    break;
  case TSTNOTMARK: /* mark */
    if (ph==2 && IS_ACTIVE && TSTBITN(cp->marks,simdinstr.arg1))
      DEACTIVATE;
    break;
  case SETMARK: /* mark */
    if (ph==2 && IS_ACTIVE)
      SETBITN(cp->marks,simdinstr.arg1);
    break;
  case CLRMARK: /* mark */
    if (ph==2 && IS_ACTIVE)
      CLRBITN(cp->marks,simdinstr.arg1);
    break;
  case CONST: /* val */
    if (ph==2 && IS_ACTIVE)
      cp->CELLACC = simdinstr.arg1;
```

```
  break;
case CLEAR: /* reg */
  if (ph==2 && IS_ACTIVE)
    cp->reg[simdinstr.arg1] = 0;
  break;
case EQ: /* reg reg */
  if (ph==2 && IS_ACTIVE &&
      !((cp->reg[simdinstr.arg1])==(cp->reg[simdinstr.arg2])))
    DEACTIVATE;
  break;
case NEQ: /* reg reg */
  if (ph==2 && IS_ACTIVE &&
      !((cp->reg[simdinstr.arg1])!=(cp->reg[simdinstr.arg2])))
    DEACTIVATE;
  break;
case GT: /* reg reg */
  if (ph==2 && IS_ACTIVE &&
      !((cp->reg[simdinstr.arg1])>(cp->reg[simdinstr.arg2])))
    DEACTIVATE;
  break;
case LE: /* reg reg */
  if (ph==2 && IS_ACTIVE &&
      !((cp->reg[simdinstr.arg1])<=(cp->reg[simdinstr.arg2])))
    DEACTIVATE;
  break;
case LT: /* reg reg */
  if (ph==2 && IS_ACTIVE &&
      !((cp->reg[simdinstr.arg1])<(cp->reg[simdinstr.arg2])))
    DEACTIVATE;
  break;
case GE: /* reg reg */
  if (ph==2 && IS_ACTIVE &&
      !((cp->reg[simdinstr.arg1])>=(cp->reg[simdinstr.arg2])))
    DEACTIVATE;
  break;
case TSTZERO: /* reg */
  if (ph==2 && IS_ACTIVE && !((cp->reg[simdinstr.arg1])==0))
    DEACTIVATE;
  break;
case TSTNZERO: /* reg */
  if (ph==2 && IS_ACTIVE && ((cp->reg[simdinstr.arg1])==0))
    DEACTIVATE;
  break;
case MOVE:
  if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] = cp->reg[simdinstr.arg2];
  break;
case ADD: /* treg, sreg */
  if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] += cp->reg[simdinstr.arg2];
  break;
case SUB: /* treg, sreg */
  if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] -= cp->reg[simdinstr.arg2];
```

```
    break;
  case LOGNOT: /* reg */
    if (ph==2 && IS_ACTIVE)
        cp->reg[simdinstr.arg1] = ~cp->reg[simdinstr.arg1];
    break;
  case LOGAND: /* treg, sreg */
    if (ph==2 && IS_ACTIVE)
        cp->reg[simdinstr.arg1] &= cp->reg[simdinstr.arg2];
    break;
  case LOGIOR: /* treg, sreg */
    if (ph==2 && IS_ACTIVE)
        cp->reg[simdinstr.arg1] |= cp->reg[simdinstr.arg2];
    break;
  case LOGXOR: /* treg sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] ^= cp->reg[simdinstr.arg2];
    break;
  case ROTATEL: /* treg sreg */
    if (ph==2 && IS_ACTIVE) {
      /* This is specifically a 16 bit operation and doesn't use
         a condition flag */
      val = cp->reg[simdinstr.arg2];
      val = ((val << 1) & 0xffff) + (val & 0x8000 ? 1 : 0);
      cp->reg[simdinstr.arg1] = val;
      }
    break;
  case ROTATER: /* treg sreg */
    if (ph==2 && IS_ACTIVE) {
      /* This is specifically a 16 bit operation and doesn't use
         a condition flag */
      val = cp->reg[simdinstr.arg2];
      val = ((val >> 1) & 0x7fff) + (val & 1 ? 0x8000 : 0);
      cp->reg[simdinstr.arg1] = val;
      }
    break;
  case SHIFTL: /* treg sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] = (cp->reg[simdinstr.arg2]) << 1;
    break;
  case SHIFTR: /* treg sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] = ((unsigned int)(cp->reg[simdinstr.arg2])) >> 1;
    break;
  case SHIFTRA: /* treg sreg */
    if (ph==2 && IS_ACTIVE)
      /* This is probably not very portable, but produdces the
         desired effect on a SPARC */
      cp->reg[simdinstr.arg1] = (cp->reg[simdinstr.arg2]) >> 1;
    break;
```

The following operations make key use of the PORTNO assignment for cells which allocates a unique wire to each cell on a port bus.

— *If mark is set on pointed at cell, stay active.*

```
case TSTMARKPTR: /* mark, source-ptr */
  /* Note: the following is done by all cells */
  if (ph==1) {
    if (IS_VALID && TSTBITN(cp->marks,simdinstr.arg1)) {
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++) SETBIT(cp->ports[r]->portval,val);
    }
  }
  else { /* ph==2 */
    if (IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg2],VALIDADDR(r)) {
        tile = TILE(r);
        dir = WHICHDIR(TILE(cp->self),tile);
        if (dir == UNDEF) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
            if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          val = PORTNO(r);
          if (!TSTBITN(cp->ports[dir]->portval,val))
            DEACTIVATE;
        }
      } else DEACTIVATE;
    }
  }
  break;
```

— If mark is clear on pointed at cell, stay active.

```
case TSTNOTMARKPTR: /* mark, source-ptr */
  /* Almost identical to the above */
  /* Note: the following is done by all cells */
  if (ph==1) {
    if (IS_VALID && TSTBITN(cp->marks,simdinstr.arg1)) {
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++) SETBIT(cp->ports[r]->portval,val);
    }
  }
  else { /* ph==2 */
    if (IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg2],VALIDADDR(r)) {
        tile = TILE(r);
        dir = WHICHDIR(TILE(cp->self),tile);
        if (dir == UNDEF) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
            if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          val = PORTNO(r);
          if (TSTBITN(cp->ports[dir]->portval,val))  — No !.
            DEACTIVATE;
        }
      } else DEACTIVATE;
    }
  }
  break;
```

```
   — Set mark on pointed at cell.
case SETMARKPTR: /* mark, target-ptr */
  if (ph==1) {
    if (IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg2],VALIDADDR(r)) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
        if (UNDEF == dir) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
            if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
        }
      } else DEACTIVATE;
    }
  } else { /* ph==2 */
    if (IS_VALID) {
    val = MSKBITN(simdinstr.arg1);
    if (!(TSTBIT(cp->marks,val))) {
      mask = val;
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++)
        if (TSTBIT(cp->ports[r]->portval,val)) {
          SETBIT(cp->marks,mask);
          break;
        }
    }
    }
  }
  break;
   — Clear mark on pointed at cell.
case CLRMARKPTR: /* mark, target-ptr */
  /* Almost identical to the above */
  if (ph==1) {
    if (IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg2],VALIDADDR(r)) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
        if (UNDEF == dir) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
            if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
        }
      } else DEACTIVATE;
    }
  } else { /* ph==2 */
    if (IS_VALID) {
    val = MSKBITN(simdinstr.arg1);
    if (TSTBIT(cp->marks,val)) {  — No !.
      mask = val;
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++)
        if (TSTBIT(cp->ports[r]->portval,val)) {
```

```
          CLRBIT(cp->marks,mask);  — Versus SETBIT.
          break;
        }
    }
    }
  }
  break;
```

The following group consists of port arbitration instructions, and fetch/store variants.

```
   — Arbitrate on specified port.
case ARBPORT: /* dir */
  if (IS_ACTIVE) {
    dir = simdinstr.arg1;
    if (ph==1) {
      if (!VALIDDIR(dir)) {
        CLRBIT(cp->state,STATE_TRYING); /*newer*/
        DEACTIVATE;
      } else
        /* This requires that the ports be cleared to start with */
        SETBITN(cp->ports[dir]->portval,cp->portno);
    } else { /* ph ==2 */
      /* lower numbered bits have higher priority */
      if ((cp->ports[dir]->portval) & LOWERBITS(cp->portno)) DEACTIVATE;
      else
        CLRBIT(cp->state,STATE_TRYING); /*newer*/
    }
  }
  break;
   — Arbitrate on port given by pointer.
case ARBPORTPTR: /* reg */
  if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
    tile = TILE(r);
    dir = WHICHDIR(TILE(cp->self),tile);
    if (ph==1) {
      if (dir == UNDEF) {
        if (simdinstr.arg1==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        if (simdinstr.arg1==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
        CLRBIT(cp->state,STATE_TRYING); /*newer*/
        DEACTIVATE;
      } else
        SETBITN(cp->ports[dir]->portval,cp->portno);
    } else { /* ph ==2 */
      if ((cp->ports[dir]->portval) & LOWERBITS(cp->portno)) DEACTIVATE;
      else
        CLRBIT(cp->state,STATE_TRYING); /*newer*/
    }
  }
  break;
  — Assumes arbitration.
  —This should be split into two instructions.
case STORE: /* target-ptr, treg, sreg */
  /* originally written using action between ph==1 and ph==2 */
  if (ph==1) {
```

```
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      tile = TILE(r);
      dir = WHICHDIR(TILE(cp->self),tile);
      if (dir == UNDEF) DEACTIVATE;
      else {
        cp->ports[dir]->portaddr = r;
        cp->ports[dir]->portval = cp->reg[simdinstr.arg3];
      }
    }
  } else { /* ph==2 */
    /* Note: the following is done by all VALID cells */
    if (IS_VALID) {
    val = cp->self;
    /* Check only one? */
    for (r=0; r<PORTCNT; r++)
      if (cp->ports[r]->portaddr == val)
        cp->reg[simdinstr.arg2] = cp->ports[r]->portval;
    }
  }
  break;
  — Assumes arbitration.
  — MAKEREQ FETCHDATA performs a fetch.
case MAKEREQ: /* ptr */
  if (ph==1) {
    cp->aux = 0; /* Note: done by all */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      tile = TILE(r);
      dir = WHICHDIR(TILE(cp->self),tile);
      if (dir == UNDEF) {
        if (simdinstr.arg1==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        if (simdinstr.arg1==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
        DEACTIVATE;
      } else
        cp->ports[dir]->portaddr = r;  /* Could xor */
    }
  } else { /* ph ==2 */
    /* Note: the following is done by all cells */
    val = cp->self;
    /* Check only one? */
    for (r=0; r<PORTCNT; r++)
      if (cp->ports[r]->portaddr == val)
        SETBITW(cp->aux,r);
  }
  break;
case FETCHDATA: /* treg, source-ptr, sreg */
  if (ph==1) {
    /* Note: the following is done by all cells */
    if (IS_VALID) {
    val = cp->aux;
    if (val != 0)
      for (r=0; r<PORTCNT; r++)
        if (TSTBITW(val,r)) {
          cp->ports[r]->portaddr = cp->self;
          cp->ports[r]->portval = cp->reg[simdinstr.arg3];
```

```
        }
      }
    } else { /* ph ==2 */
      if (IS_ACTIVE && (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
        tile = TILE(r);
        dir = WHICHDIR(TILE(cp->self),tile);
        if (dir == UNDEF) DEACTIVATE;
        else
          if (cp->ports[dir]->portaddr == r)
            cp->reg[simdinstr.arg1] = cp->ports[dir]->portval;
          else {
            printid(cp);
            printf(" fetchdata: bad data portaddr = ");
            printaddr(cp->ports[dir]->portaddr);
            printf("  portval = %d\n",cp->ports[dir]->portval);
          }
      }
    }
    break;
```

— No arbitration needed, activate by PORTNO.

```
  case STOREPH: /* portno, target-ptr, treg, sreg */
    /* originally written using action between ph==1 and ph==2 */
    if (ph==1) {
      if (IS_ACTIVE &&
          (simdinstr.arg1 == cp->portno) &&
          (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
        tile = TILE(r);
        dir = WHICHDIR(TILE(cp->self),tile);
        if (dir == UNDEF) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
          if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          cp->ports[dir]->portaddr = r;
          cp->ports[dir]->portval = cp->reg[simdinstr.arg4];
        }
      }
    } else { /* ph==2 */
      /* Note: the following is done by all cells */
      if (IS_VALID) {
      val = cp->self;
      /* Check only one? */
      for (r=0; r<PORTCNT; r++)
        if (cp->ports[r]->portaddr == val)
          cp->reg[simdinstr.arg3] = cp->ports[r]->portval;
      }
    }
    break;
```

— No arbitration needed, activate by PORTNO.

```
  case FETCHPH: /* portno, treg, source-ptr, sreg */
    if (ph==1) {
      if (IS_VALID) {
      if (simdinstr.arg1 == cp->portno) {
        val = cp->reg[simdinstr.arg4];
```

```
      for (r=0; r<PORTCNT; r++) {
        cp->ports[r]->portaddr = cp->self;
        cp->ports[r]->portval = val;
      }
    }
    }
  } else { /* ph==2 */
    if (IS_ACTIVE && (val = cp->reg[simdinstr.arg3],VALIDADDR(val))) {
      for (r=0; r<PORTCNT; r++)
        if (cp->ports[r]->portaddr == val) {
          cp->reg[simdinstr.arg2] = cp->ports[r]->portval;
          break;
        }
    }
  }
  break;
```

This initiates global feedback. If anything is active, then global feedback bit should eventually be set.

```
case SENDGLOBAL:
  if (IS_ACTIVE && ph==2) {
    globalfeedback++;
  }
  break;
```

These are the row/col instructions with associated global actions. The normal sequence is ARBTILE ARBCOL SELROW ARBROW SELCELL, and then perform global operations.

```
case ARBTILE:
  if (IS_ACTIVE) {
    if (ph==1) {
      SETBITN(cp->ports[NORTH]->portval,cp->portno);
    } else { /* ph ==2 */
      /* lower numbered bits have higher priority */
      if ((cp->ports[NORTH]->portval) & LOWERBITS(cp->portno)) DEACTIVATE;
    }
  }
  break;
case ARBCOL:
  if (IS_ACTIVE) {
    if (ph==1) {
      SETBITN(cp->col->colval,ROW(cp->self));
    } else { /* ph==2 */
      if ((cp->col->colval) & LOWERBITS(ROW(cp->self))) DEACTIVATE;
        else cp->col->colowner = cp->self;
    }
  }
  break;
case SELROW: /* target */
  if (ph==2 && IS_ACTIVE) {
    if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
      if (cp->col->colowner == cp->self)
        cp->col->colval = ROW(r);
      else {
```

```
        printid(cp);
        printf(" selrow bad active cell\n");
        DEACTIVATE;
      }
    } else DEACTIVATE;
  }
  break;
case ARBROW: /* target */
  if (ph==2 && IS_ACTIVE) {
    if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
      if (cp->col->colowner == cp->self) {
        if (TSTBIT(cp->col->colval,FAILFLAG))
          DEACTIVATE;
      } else {
        printid(cp);
        printf(" arbrow bad active cell\n");
        DEACTIVATE;
      }
    } else DEACTIVATE;
  }
  break;
case SELCELL: /* ptr */
  if (IS_ACTIVE && ph==2) {
    if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self) {
        printid(cp);
        printf(" selcell: not owner\n");
        printcell(cp);
        printaddr(cp->col->colowner); printf("\n");
        printaddr(row[ROW(r)].rowowner); printf("\n");
        printcelladdr(row[ROW(r)].rowowner);
      } else {
        CELLAT(r).aux = cp->self;
      }
    } else DEACTIVATE;
  }
  break;
case FETCHGLBL: /* target source-ptr reg */
  if (IS_ACTIVE && (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
    if (cp->col->colowner != cp->self ||
        row[ROW(r)].rowowner != cp->self ||
        CELLAT(r).aux != cp->self) {
      if (ph==1) {
        printid(cp);
        printf(" fetchglbl: not owner\n"); }
    } else {
      if (ph==2) {
        /* no cell arbitration is necessary */
        /* only really use the number of the cell from the pointer */
        cp->reg[simdinstr.arg1] = CELLAT(r).reg[simdinstr.arg3];
      }
    }
  }
```

```
  break;
case STOREGLBL: /* target-ptr reg source */
  /* as usual very similar to the above */
  if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
    if (cp->col->colowner != cp->self ||
        row[ROW(r)].rowowner != cp->self ||
        CELLAT(r).aux != cp->self) {
      if (ph==1) {
        printid(cp);
        printf(" storeglbl: not owner\n"); }
    } else {
      if (ph==2) {
        /* no cell arbitration is necessary */
        /* if (verbose) {
          printf("storeglbl: ");
          printaddr(r);
          printf(" %d ",simdinstr.arg2);
          printf(" <- %d\n",cp->reg[simdinstr.arg3]); } */
        CELLAT(r).reg[simdinstr.arg2] = cp->reg[simdinstr.arg3];
      }
    }
  }
  break;
case TSTMARKGLBL: /* source-ptr mark (changed order) */
  /* as usual very similar to the above */
  if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
    if (cp->col->colowner != cp->self ||
        row[ROW(r)].rowowner != cp->self ||
        CELLAT(r).aux != cp->self) {
      if (ph==1) {
        printid(cp);
        printf(" tstmarkglbl: not owner\n"); }
    } else {
      if (ph==2) {
        /* no cell arbitration is necessary */
        if (!TSTBITW(CELLAT(r).marks,simdinstr.arg2))
          DEACTIVATE;
      }
    }
  }
  break;
case TSTNOTMARKGLBL: /* source-ptr mark */
  /* as usual very similar to the above */
  if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
    if (cp->col->colowner != cp->self ||
        row[ROW(r)].rowowner != cp->self ||
        CELLAT(r).aux != cp->self) {
      if (ph==1) {
        printid(cp);
        printf(" tstnotmarkglbl: not owner\n"); }
    } else {
      if (ph==2) {
        /* no cell arbitration is necessary */
        if (TSTBITW(CELLAT(r).marks,simdinstr.arg2))
```

```
          DEACTIVATE;
        }
      }
    }
    break;
  case SETMARKGLBL: /* target-ptr mark */
    /* as usual very similar to the above */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self ||
          CELLAT(r).aux != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" setmarkglbl: not owner "); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          SETBITN(CELLAT(r).marks,simdinstr.arg2);
        }
      }
    }
    break;
  case CLRMARKGLBL: /* target-ptr mark */
    /* as usual very similar to the above */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self ||
          CELLAT(r).aux != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" clrmarkglbl: not owner "); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          CLRBITN(CELLAT(r).marks,simdinstr.arg2);
        }
      }
    }
    break;
  case INCRGLBL: /* val, target-ptr */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" incrglbl: not owner\n"); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          CELLAT(r).refc += simdinstr.arg1;
        }
      }
    }
    break;
```

This group allows general access to the bits of the state register, but is probably not how this state testing should be implemented.

```
/* Note: the following work with bit masks */
case TSTSTATE: /* flags */
  if (ph==2 && IS_ACTIVE && !TSTBIT(cp->state,simdinstr.arg1))
    DEACTIVATE;
  break;
case TSTNOTSTATE: /* flags */
  if (ph==2 && IS_ACTIVE && TSTBIT(cp->state,simdinstr.arg1))
    DEACTIVATE;
  break;
case SETSTATE: /* flags */
  if (ph==2 && IS_ACTIVE)
    SETBIT(cp->state,simdinstr.arg1);
  break;
case CLRSTATE: /* flags */
  if (ph==2 && IS_ACTIVE)
    CLRBIT(cp->state,simdinstr.arg1);
  break;
```

Here is another group of relatively simple instructions which are tests on addresses/pointers and the reference count register.

```
case TSTLOCAL: /* ptr */
  if (ph==2 && IS_ACTIVE) {
    if (!((r = cp->reg[simdinstr.arg1],VALIDADDR(r)) &&
          UNDEF != WHICHDIR(TILE(cp->self),TILE(r))))
      /* deactivate if not a valid address or not local */
      DEACTIVATE;
  }
  break;
case TSTREMOTE: /* ptr */
  if (ph==2 && IS_ACTIVE) {
    if (!((r = cp->reg[simdinstr.arg1],VALIDADDR(r)) &&
          UNDEF == WHICHDIR(TILE(cp->self),TILE(r))))
      /* deactivate if not a valid address or local */
      DEACTIVATE;
  }
  break;
case TSTADDR: /* ptr */
  if (ph==2 && IS_ACTIVE && !VALIDADDR(cp->reg[simdinstr.arg1]))
      /* deactivate if actually is a valid address */
      DEACTIVATE;
  break;
case TSTNOTADDR: /* ptr */
  if (ph==2 && IS_ACTIVE && VALIDADDR(cp->reg[simdinstr.arg1]))
      /* deactivate if actually is a valid address */
      DEACTIVATE;
  break;
case TSTUNUSED:
  if (ph==2 && IS_ACTIVE && 0 != cp->refc)
      DEACTIVATE;
  break;
```

```
case TSTUNSHARED:
  if (ph==2 && IS_ACTIVE && 1 != cp->refc)
      DEACTIVATE;
  break;
case TSTINUSE:
  if (ph==2 && IS_ACTIVE && cp->refc <= 0)
      DEACTIVATE;
  break;
```

These were designed for use with explicit port arbitration (**ARBPORT**, **ARBPORTPTR**), allowing a relatively tight "repeat until all done" loop.

```
case SETTRYING:
  if (ph==2) {
    if (IS_ACTIVE) {
      SETBIT(cp->state,STATE_TRYING);
      globalfeedback++;
    }
    else CLRBIT(cp->state,STATE_TRYING); /* When clear? */
  }
  break;
case INITTRYING:
  if (ph==2) {
    if (TSTALLBIT(st,STATE_VALID|STATE_TRYING)) {
      ACTIVATE;
      globalfeedback++;
    } else {
      DEACTIVATE;
    }
  }
  break;
case CLRTRYING:
  if (ph==2 && IS_ACTIVE) {
    CLRBIT(cp->state,STATE_TRYING);
  }
  break;
```

These are reference count operations (there is also a global version).

```
case INCR: /* val */
  if (IS_ACTIVE && ph==2) cp->refc += simdinstr.arg1;
  break;
case INCRPTR1: /* val,target-ptr */
case INCRPTR2:
  if (ph==1) {
    if (IS_ACTIVE &&
        (val = NUM(cp->self),(op==INCRPTR1 ? 0<=val && val<=1 :
                              2<=val && val<=3)) &&
        (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
      dir = WHICHDIR(TILE(cp->self),TILE(r));
      if (dir == UNDEF) {
        if (simdinstr.arg1==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        if (simdinstr.arg1==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
        DEACTIVATE;
```

```
      } else {
        SETBIT(cp->ports[dir]->portval,
               ((val==0||val==2)? MSKBITW(PORTNO(r)) :
                MSKBITW(PORTNO(r)+PORTNOSHIFT)));
      }
    }
  } else { /* ph==2 */
    /* IS_VALID ?? */
    val = MSKBITW(cp->portno) | MSKBITW(cp->portno+PORTNOSHIFT);
    for (r=0; r<PORTCNT; r++)
      if (TSTBIT(cp->ports[r]->portval,val)) {
        if (TSTBIT(cp->ports[r]->portval,MSKBITW(cp->portno)))
            cp->refc += simdinstr.arg1;
        if (TSTBIT(cp->ports[r]->portval,MSKBITW(cp->portno+PORTNOSHIFT)))
            cp->refc += simdinstr.arg1;
      }
  }
  break;
```

These are the allocation instructions. The normal idiom is **ALLOCREQ AVAIL1 AVAIL2 ALLOCPTR**. This idiom will leave a pointer to a free cell in the specified register (or 0, null pointer, if fails), but the cell is still not allocated. **ALLOCPTR** is used to actually allocate the cell. Many of the details could be changed and improved here.

```
case ALLOCREQ:
  if (ph==1) {
    cp->aux = 0;
    if (IS_ACTIVE) {
      val = MSKBITW(cp->portno);
      for (r=0; r<PORTCNT; r++)
        SETBIT(cp->ports[r]->portval,val);
    }
  } else { /* ph==2 */
    if (!TSTBIT(cp->state,STATE_VALID)) {
      val = 0;
      mask = PORTNOCASE(cp->portno);
      for (r=0; r<PORTCNT; r++)
        val = (val << CELLSPERTILE) |
          PORTNOSEL(mask,cp->ports[r]->portval);
      cp->aux = val; /* Could use ACC instead */
    }
  }
  break;
case AVAIL1:  /* reg */
case AVAIL2:
  if (ph==1) {
    if (cp->aux &&
        (val = NUM(cp->self),(op==AVAIL1 ? 0<=val && val<=1 :
                              2<=val && val<=3))) {
      mask = val;
      val = LOWESTBIT(cp->aux);
      for (r=PORTCNT-1; 0<=r; r--) {
        if (val & LOWERBITS(CELLSPERTILE)) {
          val = val & LOWERBITS(CELLSPERTILE);
```

```
          val = PORTNOREPLY(cp->portno,val);
          cp->ports[r]->portval |=
            ((mask==0 || mask==2) ? val : val << PORTNOSHIFT);
          break;
        }
        val >>= CELLSPERTILE;
      }
      cp->aux = 0;
    }
  } else { /* ph==2 */
    if (IS_ACTIVE) {
      if (op==AVAIL1) cp->reg[simdinstr.arg1] = 0; /*???*/
      val = MSKBITN(cp->portno) | MSKBITN(cp->portno+PORTNOSHIFT);
      if (!(op==AVAIL2 && cp->reg[simdinstr.arg1]!=0)) {
        for (r=0; r<PORTCNT; r++)
          if (TSTBIT(cp->ports[r]->portval,val)) {
            if (TSTBIT(cp->ports[r]->portval,
                       MSKBITN(cp->portno))) {
              if (op==AVAIL1) a = 0; else a = 2;
            }
            else
            if (TSTBIT(cp->ports[r]->portval,
                       MSKBITN(cp->portno+PORTNOSHIFT))) {
              if (op==AVAIL1) a = 1; else a = 3;
            }
            val = cp->self;
            switch (r) {
            case EAST:  val = ADDR(ROW(val),COL(val)+1,a); break;
            case NORTH: val = ADDR(ROW(val)-1,COL(val),a); break;
            case WEST:  val = ADDR(ROW(val),COL(val)-1,a); break;
            case SOUTH: val = ADDR(ROW(val)+1,COL(val),a); break;
            }
            cp->reg[simdinstr.arg1] = val;
            break;
          }
      }
      if (op==AVAIL2 && cp->reg[simdinstr.arg1] == 0) DEACTIVATE;
    }
  }
  break;
case ALLOCPTR: /* target-ptr */
  if (ph==1) {
    if (IS_ACTIVE)
      if ((r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
        SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
      } else DEACTIVATE;
  } else { /* ph==2 */
    /* Note: the following is done by all cells */
    if (!TSTBIT(cp->state,STATE_VALID)) {
      cp->refc = 0;
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++)
        if (TSTBIT(cp->ports[r]->portval,val)) {
```

```
          SETBIT(cp->state,STATE_VALID);
          cp->refc++; /* should really just do once */
        }

    }
  }
  break;
```

These are the commit instructions.

```
case COMMIT:
  /* failure flag? */
  if (ph==2 && IS_ACTIVE) {
    for (r=0; r<REGNUM; r++) {
      val = cp->reg[r];
      cp->reg[r] = cp->reg[r+REGNUM]; /*??*/
      cp->reg[r+REGNUM] = val;
    }
    /* This should result in eventual deletion of next stuff */
    SETBIT(cp->state,STATE_COMMIT);
  }
  break;
case COMMITMARK: /* mark */
  if (ph==2 && IS_ACTIVE) {
    if (TSTBITW(cp->marks,simdinstr.arg1)) {
      for (r=0; r<REGNUM; r++) {
        val = cp->reg[r];
        cp->reg[r] = cp->reg[r+REGNUM]; /*??*/
        cp->reg[r+REGNUM] = val;
      }
    }
    SETBIT(cp->state,STATE_COMMIT);
  }
  break;
case SWAP:
  if (ph==2 && IS_ACTIVE) {
    for (r=0; r<REGNUM; r++) {
      val = cp->reg[r];
      cp->reg[r] = cp->reg[r+REGNUM]; /*??*/
      cp->reg[r+REGNUM] = val;
    }
  }
  break;
case SETCOMMIT:
  if (ph==2 && IS_ACTIVE) {
    SETBIT(cp->state,STATE_COMMIT);
  }
  break;
```

Miscellaneous other intructions:

— In real ensemble, expand to code sequence.

```
case DELETE:
  if (ph==2 && IS_ACTIVE) {
```

```
    cp->marks = 0;
    cp->refc = 0;
    cp->state = 0;
    cp->aux = 0;
    for (r=0; r<REGCNT; r++) cp->reg[r] = 0;
  }
  break;
   — Initiate relocation if non-local.
case RELOCREQ: /* ptr */
  if (ph==2 && IS_ACTIVE) {
    if ((simdinstr.arg1 == LEFT || simdinstr.arg1 == RIGHT) &&
        (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) &&
        UNDEF == WHICHDIR(TILE(cp->self),TILE(r))) {
      if (simdinstr.arg1 == LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        SETBIT(cp->state,STATE_REMOTE_RIGHT);
      DEACTIVATE;
    }
  }
  break;
case SETRANDOM: /* reg */
  if (ph==2 && IS_ACTIVE) {
    cp->reg[simdinstr.arg1] = randoom();  — Own random function.
  }
  break;
   — May not be usable because of some IS_VALID tests.
case INITALL:
  if (ph==2) ACTIVATE;
  break;
case TSTBLACK:
  if (IS_ACTIVE && ph==2)
    if (PORTNOCASE(cp->portno))
      DEACTIVATE;
  break;
case TSTRED:
  if (IS_ACTIVE && ph==2)
    if (!PORTNOCASE(cp->portno))
      DEACTIVATE;
  break;
case TSTCLASS: /* class -- one of 5 cases */
  if (IS_ACTIVE && ph==2) {
    val = cp->self;
    if (!(simdinstr.arg1 == ((COL(val)+3*ROW(val))%5)))
      DEACTIVATE;
  }
  break;
case TSTCELLNUM: /* cellnum */
  if (IS_ACTIVE && ph==2) {
    val = cp->self;
    if (!(simdinstr.arg1 == NUM(val)))
      DEACTIVATE;
  }
  break;
case CELLARBPTR: /* target-ptr */
  if (IS_ACTIVE) {
```

```
    if (ph==1) {
      if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
        SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
      }
    } else { /* ph==2 */
      if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
        if (!TSTBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r))))
          DEACTIVATE;
      }
    }
    }
    break;
```

The action for an illegal op code is:

```
  default:
    printf("Illegal op code %d\n",op);
    fflush(stdout);
    exit(-1);
  }
}
```

The following routines are used with the SIMD instruction **SENDGLOBAL** to simulate the global feedback mechanism.

```
/* special simd operations */
resetglob()
{
  globalfeedback = 0;
}

int
testglob()
{
  return globalfeedback;
}
```

C Code of the Ensemble Simulator and of Several Benchmarks Run on it

```
/*
-----------------------------------------------------------------
        RRM "RISC" ensemble simulator in C

        Still using "SIMD code" is "real code" approach.
        Early draft: not worrying about efficiency at all.
        Revision based on various discussions.
        Goal: stricter RISC approach (e.g. simplified fetch/store).

        Typed values in registers?  Specifically pointer vs number?
-----------------------------------------------------------------
*/

/* @@
Haven't really thought about FAILURE bit stuff.
portno stuff not really handled very cleanly
*/

#include <stdio.h>
#include "def.h"
#include "risc.h"

/* ------------------------------------------------------------------ */
/* global variables */
int size;

int clock; /* Note: not phase number */

int globalfeedback;

simdbus simdinstr;

portbus port[PORTNUM];

colbus col[COLNUM];
rowbus row[ROWNUM];

cellstate cell[CELLNUM];

int recordinstrfreq = 1;
int instrfreq[OPCODENUM];

int randomstate;


/* ------------------------------------------------------------------ */
/* other globals */

int verbose;
int showinstrs;

char *opnames[OPCODENUM] = {
  "nop", "init", "erase", "tstmalmrk", "tstnotmark", "setmark",
"CLRMARK", "CONST", "CLEAR", "EQ", "NEQ", "GT", "LE", "LT", "ge", "move",
"add", "sub", "lognot", "logand", "logior", "arbport", "arbportptr", "fetch",
"store", "tstmarkptr", "tstnotmarkptr", "setmarkptr", "clrmarkptr", "initall",
"incrptr", "incr", "setcommitlock", "tstnotcommitlock", "setdeletelock",
"tstnotdeletelock", "tstblack", "tstred", "tstclass", "tstcellnum",
"sendglobal", "cellarbptr", "findfree", "allocptr", "arbcol", "selrow",
"arbrow", "fetchglbl", "storeglbl", "tstmarkglbl", "tstnotmarkglbl",
"setmarkglbl", "clrmarkglbl", "findfreedir", "commit", "tststate",
"tstnotstate", "setstate", "clrstate", "tstlocal", "tstremote", "tstaddr",
"tstnotaddr", "tstunused", "tstunshared", "makereq", "fetchdata", "settrying",
"inittrying", "incrptr1", "incrptr2", "allocreq", "avail1", "avail2",
"relocreq", "storepn", "fetchpn", "incrglbl", "clrtrying", "delete",
"commitmark", "swap", "tstinuse", "setcommit", "tstzero", "tstnzero",
"selcell", "arbtile", "setrandom", "logxor", "rotatel", "rotater", "shiftl",
"shiftr", "shiftra", "alloc"
  };

char name[50],init[50],prog[50];
  FILE *fp;

/* ------------------------------------------------------------------ */
main(argc,argv,env)
int argc;
char **argv,**env;
{

  int i;

  verbose = (NULL != getenv("VERBOSE"));
  randomstate = 232323;
  show_ensemble_fn = show_ensemble_std;
  /* initialize structures */
  ensemble_init();
  i=argc;
  size = 4;

     if(i) {sscanf(argv[0],"%s",name);  i--;}
     strcpy(init,strcat(name,".init"));
     if(i)
     {
         sscanf(argv[1],"%s",prog);
         i--;
         if((fp=fopen(prog,"r"))==NULL)
         {
            printf("cannot access %s ..\n",init);
            exit(1);
         }
     }
     else fp=stdin;
/*
  if(argc>2)
  {
     if((fp=fopen(init,"r"))==NULL)
     {
        printf("cannot access %s ..\n",init);
        exit(1);
     }
     for(i=0;i<CELLNUM;i++) fread(&cell[i],sizeof(cell[0]),1,fp);
  }
  else
  {
      term_build();
      if((fp=fopen(init,"w"))==NULL)
      {
            printf("cannot access %s ..\n",init);
            exit(1);
      }
      for(i=0;i<CELLNUM;i++) fwrite(&cell[i],sizeof(cell[0]),1,fp);

  }
 */
  /* execute SIMD code */
  simd_execute();
}
```

```
/* ------------------------------------------------------------------ */
simd(op)
int op;
{
  simdinstr.op = op;
  simdinstr.arg1 = UNDEF;
  simdinstr.arg2 = UNDEF;
  simdinstr.arg3 = UNDEF;
  simdinstr.arg4 = UNDEF;
  simdstep();
}
simd1(op,arg)
int op,arg;
{
  simdinstr.op = op;
  simdinstr.arg1 = arg;
  simdinstr.arg2 = UNDEF;
  simdinstr.arg3 = UNDEF;
  simdinstr.arg4 = UNDEF;
  simdstep();
}
simd2(op,arg1,arg2)
int op,arg1,arg2;
{
  simdinstr.op = op;
  simdinstr.arg1 = arg1;
  simdinstr.arg2 = arg2;
  simdinstr.arg3 = UNDEF;
  simdinstr.arg4 = UNDEF;
  simdstep();
}
simd3(op,arg1,arg2,arg3)
int op,arg1,arg2,arg3;
{
  simdinstr.op = op;
  simdinstr.arg1 = arg1;
  simdinstr.arg2 = arg2;
  simdinstr.arg3 = arg3;
  simdinstr.arg4 = UNDEF;
  simdstep();
}
simd4(op,arg1,arg2,arg3,arg4)
int op,arg1,arg2,arg3,arg4;
{
  simdinstr.op = op;
  simdinstr.arg1 = arg1;
  simdinstr.arg2 = arg2;
  simdinstr.arg3 = arg3;
  simdinstr.arg4 = arg4;
  simdstep();
}

simdstep()
{
  register int i,op,p,flg,val;

  clock++;

  op = simdinstr.op;
  if (recordinstrfreq) {
    if (op != INIT) instrfreq[op]++;
  }
  if (showinstrs) {
    printf("%d: ",clock);
    printop(); printf("\n"); fflush(stdout); }

  if (op != NOP) {
  /* Actually there seem to be cases here */
  /* arbcol, arbrow are actions over different entities */

  /* @@ could improve on this */
  if (op==ARBCOL)
    for (i=0; i<COLNUM; i++) { col[i].colval = 0; col[i].colowner = 0; }
  else
  if (op==SELROW)
    for (i=0; i<COLNUM; i++) col[i].colval = UNDEF;
  else
  if (op==SELCELL)
    for (i=0; i<CELLNUM; i++) cell[i].aux = 0;

  for (i=0; i<PORTNUM; i++) port[i].portval = 0;
  for (i=0; i<PORTNUM; i++) port[i].portaddr = UNDEF;

  /* if (verbose) printf("phase1\n"); */

  simdinstr.ph = 1;
  for (i=0; i<CELLNUM; i++)
    simdaction(&cell[i]);

  /* if (verbose) printf("port\n"); */
  /* In the following could put checks to see if bus already non-UNDEF
     which would indicate an error in use; this also ignores issues
     of cell arb */
  if (op == FETCH) {
    for (i=0; i<PORTNUM; i++)
      if (port[i].portaddr != UNDEF)
        port[i].portval = CELLAT(port[i].portaddr).reg[simdinstr.arg3];
  }
  else
  if (op == CELLARBPTR) {
    for (i=0; i<CELLNUM; i++) {
      flg = 0;
      val = MSKBITN(cell[i].portno);
      for (p=0; p<PORTCNT; p++) {
        if (TSTBIT(cell[i].ports[p]->portval,val)) {
          if (flg) CLRBIT(cell[i].ports[p]->portval,val);
          flg = 1;
        }
      }
    }
  }
  else
  if (op == ARBROW) {
    /* @@@ this could be much improved(!!) */
    /*@@
    for (i=0; i<ROWNUM; i++) {
      flg = 1;
      for (p=0; p<COLNUM; p++) {
        if (col[p].colval == i) {
          if (flg) row[i].rowowner = col[p].colowner;
            else SETBIT(col[p].colval,FAILFLAG);
          flg = 0;
        }
      }
    }
    */
    for (i=0; i<ROWNUM; i++) { row[i].rowval = UNDEF; row[i].rowowner = 0; }
```

```
    for (p=COLNUM-1; 0<=p; p--) {
      if (0 != col[p].colowner) {
        val = col[p].colval;
        if (val != UNDEF) {
          i = row[val].rowowner;
          if (i != 0) SETBIT(col[COL(i)].colval,FAILFLAG);
          row[val].rowowner = col[p].colowner;
        }
      }
    }
  }


  /* if (verbose) printf("phase2\n"); */
  simdinstr.ph = 2;
  for (i=0; i<CELLNUM; i++)
    simdaction(&cell[i]);
  }
}

/* ------------------------------------------------------------------- */
simdaction(cp)
register cellstate *cp;
{
  register int op,ph,st,r,val;
  int dir,tile,mask,a,flg;

  /* Do some top-level case analysis depending on whether active, etc. */
  st = cp->state;
#define DEACTIVATE CLRBIT(cp->state,STATE_ACTIVE)
#define ACTIVATE SETBIT(cp->state,STATE_ACTIVE)
  op = simdinstr.op;
  ph = simdinstr.ph;
#define IS_ACTIVE TSTBIT(st,STATE_ACTIVE)
#define IS_VALID TSTBIT(st,STATE_VALID)

  switch (op) {
  case NOP: break;
  case INIT:
    if (ph==2 && TSTBIT(st,STATE_VALID)) ACTIVATE;
    break;
  case ERASE:
    if (ph==2 && IS_ACTIVE) cp->marks = 0;
    break;
  case TSTMARK: /* mark */
    if (ph==2 && IS_ACTIVE && !TSTBITN(cp->marks,simdinstr.arg1))
      DEACTIVATE;
    break;
  case TSTNOTMARK: /* mark */
    if (ph==2 && IS_ACTIVE && TSTBITN(cp->marks,simdinstr.arg1))
      DEACTIVATE;
    break;
  case SETMARK: /* mark */
    if (ph==2 && IS_ACTIVE)
      SETBITN(cp->marks,simdinstr.arg1);
    break;
  case CLRMARK: /* mark */
    if (ph==2 && IS_ACTIVE)
      CLRBITN(cp->marks,simdinstr.arg1);
    break;
  case CONST: /* val */
    if (ph==2 && IS_ACTIVE)
      cp->CELLACC = simdinstr.arg1;
    break;
  case CLEAR: /* reg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] = 0;
    break;
  case EQ: /* reg reg */
    if (ph==2 && IS_ACTIVE &&
        !((cp->reg[simdinstr.arg1])==(cp->reg[simdinstr.arg2])))
      DEACTIVATE;
    break;
  case NEQ: /* reg reg */
    if (ph==2 && IS_ACTIVE &&
        !((cp->reg[simdinstr.arg1])!=(cp->reg[simdinstr.arg2])))
      DEACTIVATE;
    break;
  case GT: /* reg reg */
    if (ph==2 && IS_ACTIVE &&
        !((cp->reg[simdinstr.arg1])>(cp->reg[simdinstr.arg2])))
      DEACTIVATE;
    break;
  case LE: /* reg reg */
    if (ph==2 && IS_ACTIVE &&
        !((cp->reg[simdinstr.arg1])<=(cp->reg[simdinstr.arg2])))
      DEACTIVATE;
    break;
  case LT: /* reg reg */
    if (ph==2 && IS_ACTIVE &&
        !((cp->reg[simdinstr.arg1])<(cp->reg[simdinstr.arg2])))
      DEACTIVATE;
    break;
  case GE: /* reg reg */
    if (ph==2 && IS_ACTIVE &&
        !((cp->reg[simdinstr.arg1])>=(cp->reg[simdinstr.arg2])))
      DEACTIVATE;
    break;
  case MOVE:
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] = cp->reg[simdinstr.arg2];
    break;
  case ADD: /* treg, sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] += cp->reg[simdinstr.arg2];
    break;
  case SUB: /* treg, sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] -= cp->reg[simdinstr.arg2];
    break;
  case LOGNOT: /* reg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] = ~cp->reg[simdinstr.arg1];
    break;
  case LOGAND: /* treg, sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] &= cp->reg[simdinstr.arg2];
    break;
  case LOGIOR: /* treg, sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] |= cp->reg[simdinstr.arg2];
    break;
  case ARBPORT: /* dir */
    if (IS_ACTIVE) {
      dir = simdinstr.arg1;
      if (ph==1) {
        if (!VALIDDIR(dir)) {
          CLRBIT(cp->state,STATE_TRYING); /*@@ newer*/
```

```
        DEACTIVATE;
      } else
        /* This requires that the ports be cleared to start with */
        SETBITN(cp->ports[dir]->portval,cp->portno);
    } else { /* ph ==2 */
      /* lower numbered bits have higher priority */
      if ((cp->ports[dir]->portval) & LOWERBITS(cp->portno)) DEACTIVATE;
      else
        CLRBIT(cp->state,STATE_TRYING); /*@@ newer*/
    }
  }
  break;
case ARBPORTPTR: /* reg */
  if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
    tile = TILE(r);
    dir = WHICHDIR(TILE(cp->self),tile);
    if (ph==1) {
      if (dir == UNDEF) {
        if (simdinstr.arg1==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        if (simdinstr.arg1==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
        CLRBIT(cp->state,STATE_TRYING); /*@@ newer*/
        DEACTIVATE;
      } else
        SETBITN(cp->ports[dir]->portval,cp->portno);
    } else { /* ph ==2 */
      if ((cp->ports[dir]->portval) & LOWERBITS(cp->portno)) DEACTIVATE;
      else
        CLRBIT(cp->state,STATE_TRYING); /*@@ newer*/
    }
  }
  break;
case FETCH: /* treg, source-ptr, sreg */
  if (IS_ACTIVE && (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
    tile = TILE(r);
    dir = WHICHDIR(TILE(cp->self),tile);
    if (ph==1) {
      if (dir == UNDEF) {
        if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
        DEACTIVATE;
      } else
        cp->ports[dir]->portaddr = r;  /* could xor */
    } else { /* ph ==2 */
      cp->reg[simdinstr.arg1] = cp->ports[dir]->portval;
    }
  }
  break;
case STORE: /* target-ptr, treg, sreg */
  /* originally written using action between ph==1 and ph==2 */
  if (ph==1) {
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      tile = TILE(r);
      dir = WHICHDIR(TILE(cp->self),tile);
      if (dir == UNDEF) DEACTIVATE;
      else {
        cp->ports[dir]->portaddr = r;
        cp->ports[dir]->portval = cp->reg[simdinstr.arg3];
      }
    }
  } else { /* ph==2 */
    /* Note: the following is done by all VALID cells */
    if (IS_VALID) {
    val = cp->self;
    /* @@ check only one? */
    for (r=0; r<PORTCNT; r++)
      if (cp->ports[r]->portaddr == val)
        cp->reg[simdinstr.arg2] = cp->ports[r]->portval;
    }
  }
  break;
case TSTMARKPTR: /* mark, source-ptr */
  /* Note: the following is done by all cells */
  if (ph==1) {
    if (IS_VALID && TSTBITN(cp->marks,simdinstr.arg1)) {
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++) SETBIT(cp->ports[r]->portval,val);
    }
  }
  else { /* ph==2 */
    if (IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg2],VALIDADDR(r)) {
        tile = TILE(r);
        dir = WHICHDIR(TILE(cp->self),tile);
        if (dir == UNDEF) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
            if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          val = PORTNO(r);
          if (!TSTBITN(cp->ports[dir]->portval,val))
            DEACTIVATE;
        }
      } else DEACTIVATE;
    }
  }
  break;
case TSTNOTMARKPTR: /* mark, source-ptr */
  /* @@ almost identical to the above */
  /* Note: the following is done by all cells */
  if (ph==1) {
    if (IS_VALID && TSTBITN(cp->marks,simdinstr.arg1)) {
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++) SETBIT(cp->ports[r]->portval,val);
    }
  }
  else { /* ph==2 */
    if (IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg2],VALIDADDR(r)) {
        tile = TILE(r);
        dir = WHICHDIR(TILE(cp->self),tile);
        if (dir == UNDEF) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
            if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          val = PORTNO(r);
          if (TSTBITN(cp->ports[dir]->portval,val))
            DEACTIVATE;
        }
      } else DEACTIVATE;
    }
  }
  break;
case SETMARKPTR: /* mark, target-ptr */
  if (ph==1) {
    if (IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg2],VALIDADDR(r)) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
```

```
        if (UNDEF == dir) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
            if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
        }
      } else DEACTIVATE;
    }
  } else { /* ph==2 */
    if (IS_VALID) {
    val = MSKBITN(simdinstr.arg1);
    if (!(TSTBIT(cp->marks,val))) {
      mask = val;
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++)
        if (TSTBIT(cp->ports[r]->portval,val)) {
          SETBIT(cp->marks,mask);
          break;
        }
    }
    }
  }
  break;
case CLRMARKPTR: /* mark, target-ptr */
  /* @@ almost identical to the above */
  if (ph==1) {
    if (IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg2],VALIDADDR(r)) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
        if (UNDEF == dir) {
          if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
            if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else {
          SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
        }
      } else DEACTIVATE;
    }
  } else { /* ph==2 */
    if (IS_VALID) {
    val = MSKBITN(simdinstr.arg1);
    if (TSTBIT(cp->marks,val)) {
      mask = val;
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++)
        if (TSTBIT(cp->ports[r]->portval,val)) {
          CLRBIT(cp->marks,mask);
          break;
        }
    }
    }
  }
  break;
case INITALL:
  if (ph==2) ACTIVATE;
  break;
case INCRPTR: /* val,target-ptr */
  if (ph==1) {
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
      dir = WHICHDIR(TILE(cp->self),TILE(r));
      SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
    }
  } else { /* ph==2 */
    if (IS_VALID) {
    val = MSKBITN(cp->portno);
    for (r=0; r<PORTCNT; r++)
      if (TSTBIT(cp->ports[r]->portval,val)) {
        cp->refc += simdinstr.arg1;
      }
    }
  }
  break;
case INCR: /* val */
  if (IS_ACTIVE && ph==2) cp->refc += simdinstr.arg1;
  break;
case  SETCOMMITLOCK:
  if (IS_ACTIVE && ph==2)
    SETBIT(cp->state,STATE_COMMITLOCK);
  break;
case TSTNOTCOMMITLOCK:
  if (IS_ACTIVE && ph==2)
    if (TSTBIT(cp->state,STATE_COMMITLOCK))
      DEACTIVATE;
  break;
case  SETDELETELOCK:
  if (IS_ACTIVE && ph==2)
    SETBIT(cp->state,STATE_DELETELOCK);
  break;
case TSTNOTDELETELOCK:
  if (IS_ACTIVE && ph==2)
    if (TSTBIT(cp->state,STATE_DELETELOCK))
      DEACTIVATE;
  break;
case TSTBLACK:
  if (IS_ACTIVE && ph==2)
    if (PORTNOCASE(cp->portno))
      DEACTIVATE;
  break;
case TSTRED:
  if (IS_ACTIVE && ph==2)
    if (!PORTNOCASE(cp->portno))
      DEACTIVATE;
  break;
case TSTCLASS: /* class -- one of 5 cases */
  if (IS_ACTIVE && ph==2) {
    val = cp->self;
    if (!(simdinstr.arg1 == ((COL(val)+3*ROW(val))%5)))
      DEACTIVATE;
  }
  break;
/*************new**************/
case ALLOC:
  if (IS_ACTIVE && ph==2)
  {
     val=cp->self;
     if (!(simdinstr.arg1 == val)) DEACTIVATE;
     else { SETBIT(cp->state,STATE_VALID); cp->refc=1; }
  }
  break;
/*****************************/
case TSTCELLNUM: /* cellnum */
  if (IS_ACTIVE && ph==2) {
    val = cp->self;
    if (!(simdinstr.arg1 == NUM(val)))
      DEACTIVATE;
  }
  break;
```

```
  case SENDGLOBAL:
    if (IS_ACTIVE && ph==2) {
      globalfeedback++;
    }
    break;
  case CELLARBPTR: /* target-ptr */
    if (IS_ACTIVE) {
    if (ph==1) {
      if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
        SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
      }
    } else { /* ph==2 */
      if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
        dir = WHICHDIR(TILE(cp->self),TILE(r));
        if (!TSTBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r))))
          DEACTIVATE;
      }
    }
    }
    break;
  case FINDFREE: /* target */
    if (ph==1) {
      /* Note: the following is done by all cells */
      if (!TSTBIT(cp->state,STATE_VALID)) {
        val = MSKBITN(cp->portno);
        for (r=0; r<PORTCNT; r++)
          SETBIT(cp->ports[r]->portval,val);
      }
    } else { /* ph==2 */
      if (IS_ACTIVE) {
        /* Am assuming that this cell is not free */
        /* @@ This is a bit much for a poor little cell */
        cp->reg[simdinstr.arg1] = UNDEF;
        flg = PORTNOCASE(cp->portno);
        for (r=0; r<PORTCNT; r++)
          if ((val = cp->ports[r]->portval) != 0) {
            /* want to find in the "other" tile */
            /* @@ NOT SO NICE ... */
            a = bitpos(PORTNOSEL(flg,val)) & 0x3;
            val = cp->self;
            switch (r) {
            case EAST:  val = ADDR(ROW(val),COL(val)+1,a); break;
            case NORTH: val = ADDR(ROW(val)-1,COL(val),a); break;
            case WEST:  val = ADDR(ROW(val),COL(val)-1,a); break;
            case SOUTH: val = ADDR(ROW(val)+1,COL(val),a); break;
            }
            cp->reg[simdinstr.arg1] = val;
            break;
          }
      }
    }
    break;
  case ALLOCPTR: /* target-ptr */
    if (ph==1) {
      if (IS_ACTIVE)
        if ((r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
          dir = WHICHDIR(TILE(cp->self),TILE(r));
          SETBIT(cp->ports[dir]->portval,MSKBITN(PORTNO(r)));
        } else DEACTIVATE;
    } else { /* ph==2 */
      /* Note: the following is done by all cells */
      if (!TSTBIT(cp->state,STATE_VALID)) {
        cp->refc = 0;
        val = MSKBITN(cp->portno);
        for (r=0; r<PORTCNT; r++)
          if (TSTBIT(cp->ports[r]->portval,val)) {
            SETBIT(cp->state,STATE_VALID);
            cp->refc++; /* should really just do once */
          }
      }
    }
    break;
  case ARBCOL:
    if (IS_ACTIVE) {
      if (ph==1) {
        SETBITN(cp->col->colval,ROW(cp->self));
      } else { /* ph==2 */
        if ((cp->col->colval) & LOWERBITS(ROW(cp->self))) DEACTIVATE;
          else cp->col->colowner = cp->self;
      }
    }
    break;
  case SELROW: /* target */
    if (ph==2 && IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
        if (cp->col->colowner == cp->self)
          cp->col->colval = ROW(r);
        else {
          printid(cp);
          printf(" selrow bad active cell\n");
          DEACTIVATE;
        }
      } else DEACTIVATE;
    }
    break;
  case ARBROW: /* target */
    if (ph==2 && IS_ACTIVE) {
      if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
        if (cp->col->colowner == cp->self) {
          if (TSTBIT(cp->col->colval,FAILFLAG))
            DEACTIVATE;
        } else {
          printid(cp);
          printf(" arbrow bad active cell\n");
          DEACTIVATE;
        }
      } else DEACTIVATE;
    }
    break;
  case FETCHGLBL: /* target source-ptr reg */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self ||
          CELLAT(r).aux != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" fetchglbl: not owner\n"); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          /* only really use the number of the cell from the pointer */
          cp->reg[simdinstr.arg1] = CELLAT(r).reg[simdinstr.arg3];
        }
      }
    }
    break;
  case STOREGLBL: /* target-ptr reg source */
```

```
    /* as usual very similar to the above */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self ||
          CELLAT(r).aux != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" storeglbl: not owner\n"); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          /* if (verbose) {
            printf("storeglbl: ");
            printaddr(r);
            printf(" %d ",simdinstr.arg2);
            printf(" <- %d\n",cp->reg[simdinstr.arg3]); } */
          CELLAT(r).reg[simdinstr.arg2] = cp->reg[simdinstr.arg3];
        }
      }
    }
    break;
  case TSTMARKGLBL: /* source-ptr mark @@@changed order */
    /* as usual very similar to the above */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self ||
          CELLAT(r).aux != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" tstmarkglbl: not owner\n"); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          if (!TSTBITN(CELLAT(r).marks,simdinstr.arg2))
            DEACTIVATE;
        }
      }
    }
    break;
  case TSTNOTMARKGLBL: /* source-ptr mark */
    /* as usual very similar to the above */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self ||
          CELLAT(r).aux != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" tstnotmarkglbl: not owner\n"); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          if (TSTBITN(CELLAT(r).marks,simdinstr.arg2))
            DEACTIVATE;
        }
      }
    }
    break;
  case SETMARKGLBL: /* target-ptr mark */
    /* as usual very similar to the above */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self ||
          CELLAT(r).aux != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" setmarkglbl: not owner "); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          SETBITN(CELLAT(r).marks,simdinstr.arg2);
        }
      }
    }
    break;
  case CLRMARKGLBL: /* target-ptr mark */
    /* as usual very similar to the above */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self ||
          CELLAT(r).aux != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" clrmarkglbl: not owner "); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          CLRBITN(CELLAT(r).marks,simdinstr.arg2);
        }
      }
    }
    break;
  case FINDFREEDIR: /* target dir */
    if (ph==1) {
      /* Note: the following is done by all cells */
      if (!TSTBIT(cp->state,STATE_VALID)) {
        val = MSKBITN(cp->portno);
        for (r=0; r<PORTCNT; r++)
          SETBIT(cp->ports[r]->portval,val);
      }
    } else { /* ph==2 */
      if (IS_ACTIVE) {
        dir = simdinstr.arg2;
        if (VALIDDIR(dir)) {
          cp->reg[simdinstr.arg1] = UNDEF;
          flg = PORTNOCASE(cp->portno);
          if ((val = cp->ports[dir]->portval) != 0) {
            /* want to find in the "other" tile */
            /* @@ NICENESS */
            a = bitpos(PORTNOSEL(flg,val)) & 0x3;
            val = cp->self;
            switch (dir) {
            case EAST:  val = ADDR(ROW(val),COL(val)+1,a); break;
            case NORTH: val = ADDR(ROW(val)-1,COL(val),a); break;
            case WEST:  val = ADDR(ROW(val),COL(val)-1,a); break;
            case SOUTH: val = ADDR(ROW(val)+1,COL(val),a); break;
            }
            cp->reg[simdinstr.arg1] = val;
          }
        }
        else DEACTIVATE;
      }
    }
    break;
  case COMMIT:
    /* failure flag? */
    if (ph==2 && IS_ACTIVE) {
      for (r=0; r<REGNUM; r++) {
        val = cp->reg[r];
```

```
        cp->reg[r] = cp->reg[r+REGNUM]; /*@@*/
        cp->reg[r+REGNUM] = val;
      }
      /* @@ This should result in eventual deletion of next stuff */
      SETBIT(cp->state,STATE_COMMIT);
    }
    break;
  /* Note: the following work with bit masks */
  case TSTSTATE: /* flags */
    if (ph==2 && IS_ACTIVE && !TSTBIT(cp->state,simdinstr.arg1))
      DEACTIVATE;
    break;
  case TSTNOTSTATE: /* flags */
    if (ph==2 && IS_ACTIVE && TSTBIT(cp->state,simdinstr.arg1))
      DEACTIVATE;
    break;
  case SETSTATE: /* flags */
    if (ph==2 && IS_ACTIVE)
      SETBIT(cp->state,simdinstr.arg1);
    break;
  case CLRSTATE: /* flags */
    if (ph==2 && IS_ACTIVE)
      CLRBIT(cp->state,simdinstr.arg1);
    break;
  case TSTLOCAL: /* ptr */
    if (ph==2 && IS_ACTIVE) {
      if (!((r = cp->reg[simdinstr.arg1],VALIDADDR(r)) &&
            UNDEF != WHICHDIR(TILE(cp->self),TILE(r))))
        /* deactivate if not a valid address or not local */
        DEACTIVATE;
    }
    break;
  case TSTREMOTE: /* ptr */
    if (ph==2 && IS_ACTIVE) {
      if (!((r = cp->reg[simdinstr.arg1],VALIDADDR(r)) &&
            UNDEF == WHICHDIR(TILE(cp->self),TILE(r))))
        /* deactivate if not a valid address or local */
        DEACTIVATE;
    }
    break;
  case TSTADDR: /* ptr */
    if (ph==2 && IS_ACTIVE && !VALIDADDR(cp->reg[simdinstr.arg1]))
        /* deactivate if actually is a valid address */
        DEACTIVATE;
    break;
  case TSTNOTADDR: /* ptr */
    if (ph==2 && IS_ACTIVE && VALIDADDR(cp->reg[simdinstr.arg1]))
        /* deactivate if actually is a valid address */
        DEACTIVATE;
    break;
  case TSTUNUSED:
    if (ph==2 && IS_ACTIVE && 0 != cp->refc)
        DEACTIVATE;
    break;
  case TSTUNSHARED:
    if (ph==2 && IS_ACTIVE && 1 != cp->refc)
        DEACTIVATE;
    break;
  case MAKEREQ: /* ptr */
    if (ph==1) {
      cp->aux = 0; /* Note: done by all */
      if (IS_ACTIVE && (r = cp->reg[simdinstr.arg1],VALIDADDR(r))) {
        tile = TILE(r);
        dir = WHICHDIR(TILE(cp->self),tile);
        if (dir == UNDEF) {
          if (simdinstr.arg1==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
          if (simdinstr.arg1==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
          DEACTIVATE;
        } else
          cp->ports[dir]->portaddr = r;  /* @@ could xor */
      }
    } else { /* ph ==2 */
      /* Note: the following is done by all cells */
      val = cp->self;
      /* @@ check only one? */
      for (r=0; r<PORTCNT; r++)
        if (cp->ports[r]->portaddr == val)
          SETBITN(cp->aux,r);
    }
    break;
  case FETCHDATA: /* treg, source-ptr, sreg */
    if (ph==1) {
      /* Note: the following is done by all cells */
      if (IS_VALID) {
      val = cp->aux;
      if (val != 0)
        for (r=0; r<PORTCNT; r++)
          if (TSTBITN(val,r)) {
            cp->ports[r]->portaddr = cp->self;
            cp->ports[r]->portval = cp->reg[simdinstr.arg3];
          }
      }
    } else { /* ph ==2 */
      if (IS_ACTIVE && (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
        tile = TILE(r);
        dir = WHICHDIR(TILE(cp->self),tile);
        if (dir == UNDEF) DEACTIVATE;
        else
          if (cp->ports[dir]->portaddr == r)
            cp->reg[simdinstr.arg1] = cp->ports[dir]->portval;
          else {
            printid(cp);
            printf(" fetchdata: bad data portaddr = ");
            printaddr(cp->ports[dir]->portaddr);
            printf("  portval = %d\n",cp->ports[dir]->portval);
          }
      }
    }
    break;
  case SETTRYING:
    if (ph==2) {
      if (IS_ACTIVE) {
        SETBIT(cp->state,STATE_TRYING);
        globalfeedback++;
      }
      else CLRBIT(cp->state,STATE_TRYING); /* When clear? */
    }
    break;
  case INITTRYING:
    if (ph==2) {
      if (TSTALLBIT(st,STATE_VALID|STATE_TRYING)) {
        ACTIVATE;
        globalfeedback++;
      } else {
        DEACTIVATE;
      }
    }
    break;
```

```
case INCRPTR1: /* val,target-ptr */
case INCRPTR2:
  if (ph==1) {
    if (IS_ACTIVE &&
        (val = NUM(cp->self),(op==INCRPTR1 ? 0<=val && val<=1 :
                              2<=val && val<=3)) &&
        (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
      dir = WHICHDIR(TILE(cp->self),TILE(r));
      if (dir == UNDEF) {
        if (simdinstr.arg1==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        if (simdinstr.arg1==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
        DEACTIVATE;
      } else {
        SETBIT(cp->ports[dir]->portval,
               ((val==0||val==2)? MSKBITN(PORTNO(r)) :
                MSKBITN(PORTNO(r)+PORTNOSHIFT)));
      }
    }
  } else { /* ph==2 */
    /* @@ IS_VALID ?? */
    val = MSKBITN(cp->portno) | MSKBITN(cp->portno+PORTNOSHIFT);
    for (r=0; r<PORTCNT; r++)
      if (TSTBIT(cp->ports[r]->portval,val)) {
        if (TSTBIT(cp->ports[r]->portval,MSKBITN(cp->portno)))
            cp->refc += simdinstr.arg1;
        if (TSTBIT(cp->ports[r]->portval,MSKBITN(cp->portno+PORTNOSHIFT)))
            cp->refc += simdinstr.arg1;
      }
  }
  break;
case ALLOCREQ:
  if (ph==1) {
    cp->aux = 0;
    if (IS_ACTIVE) {
      val = MSKBITN(cp->portno);
      for (r=0; r<PORTCNT; r++)
        SETBIT(cp->ports[r]->portval,val);
    }
  } else { /* ph==2 */
    if (!TSTBIT(cp->state,STATE_VALID)) {
      val = 0;
      mask = PORTNOCASE(cp->portno);
      for (r=0; r<PORTCNT; r++)
        val = (val << CELLSPERTILE) |
          PORTNOSEL(mask,cp->ports[r]->portval);
      cp->aux = val; /* @@ could use ACC instead */
    }
  }
  break;
case AVAIL1:  /* reg */
case AVAIL2:
  if (ph==1) {
    if (cp->aux &&
        (val = NUM(cp->self),(op==AVAIL1 ? 0<=val && val<=1 :
                              2<=val && val<=3))) {
      mask = val;
      val = LOWESTBIT(cp->aux);
      for (r=PORTCNT-1; 0<=r; r--) {
        if (val & LOWERBITS(CELLSPERTILE)) {
          val = val & LOWERBITS(CELLSPERTILE);
          val = PORTNOREPLY(cp->portno,val);
          cp->ports[r]->portval |=
            ((mask==0 || mask==2) ? val : val << PORTNOSHIFT);
          break;
        }
        val >>= CELLSPERTILE;
      }
      cp->aux = 0;
    }
  } else { /* ph==2 */
    if (IS_ACTIVE) {
      if (op==AVAIL1) cp->reg[simdinstr.arg1] = 0; /*@@*/
      val = MSKBITN(cp->portno) | MSKBITN(cp->portno+PORTNOSHIFT);
      if (!(op==AVAIL2 && cp->reg[simdinstr.arg1]!=0)) {
        for (r=0; r<PORTCNT; r++)
          if (TSTBIT(cp->ports[r]->portval,val)) {
            if (TSTBIT(cp->ports[r]->portval,
                       MSKBITN(cp->portno))) {
              if (op==AVAIL1) a = 0; else a = 2;
            }
            else
            if (TSTBIT(cp->ports[r]->portval,
                       MSKBITN(cp->portno+PORTNOSHIFT))) {
              if (op==AVAIL1) a = 1; else a = 3;
            }
            val = cp->self;
            switch (r) {
            case EAST:  val = ADDR(ROW(val),COL(val)+1,a); break;
            case NORTH: val = ADDR(ROW(val)-1,COL(val),a); break;
            case WEST:  val = ADDR(ROW(val),COL(val)-1,a); break;
            case SOUTH: val = ADDR(ROW(val)+1,COL(val),a); break;
            }
            cp->reg[simdinstr.arg1] = val;
            break;
          }
      }
      if (op==AVAIL2 && cp->reg[simdinstr.arg1] == 0) DEACTIVATE;
    }
  }
  break;
case RELOCREQ: /* ptr */
  if (ph==2 && IS_ACTIVE) {
    if ((simdinstr.arg1 == LEFT || simdinstr.arg1 == RIGHT) &&
        (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) &&
        UNDEF == WHICHDIR(TILE(cp->self),TILE(r))) {
      if (simdinstr.arg1 == LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        SETBIT(cp->state,STATE_REMOTE_RIGHT);
      DEACTIVATE;
    }
  }
  break;
case STOREPN: /* portno, target-ptr, treg, sreg */
  /* originally written using action between ph==1 and ph==2 */
  if (ph==1) {
    if (IS_ACTIVE &&
        (simdinstr.arg1 == cp->portno) &&
        (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
      tile = TILE(r);
      dir = WHICHDIR(TILE(cp->self),tile);
      if (dir == UNDEF) {
        if (simdinstr.arg2==LEFT) SETBIT(cp->state,STATE_REMOTE_LEFT); else
        if (simdinstr.arg2==RIGHT) SETBIT(cp->state,STATE_REMOTE_RIGHT);
        DEACTIVATE;
      } else {
        cp->ports[dir]->portaddr = r;
        cp->ports[dir]->portval = cp->reg[simdinstr.arg4];
      }
    }
```

```
    } else { /* ph==2 */
      /* Note: the following is done by all cells */
      if (IS_VALID) {
      val = cp->self;
      /* @@ check only one? */
      for (r=0; r<PORTCNT; r++)
        if (cp->ports[r]->portaddr == val)
          cp->reg[simdinstr.arg3] = cp->ports[r]->portval;
      }
    }
    break;
  case FETCHPN: /* portno, treg, source-ptr, sreg */
    if (ph==1) {
      if (IS_VALID) {
      if (simdinstr.arg1 == cp->portno) {
        val = cp->reg[simdinstr.arg4];
        for (r=0; r<PORTCNT; r++) {
          cp->ports[r]->portaddr = cp->self;
          cp->ports[r]->portval = val;
        }
      }
      }
    } else { /* ph==2 */
      if (IS_ACTIVE && (val = cp->reg[simdinstr.arg3],VALIDADDR(val))) {
        for (r=0; r<PORTCNT; r++)
          if (cp->ports[r]->portaddr == val) {
            cp->reg[simdinstr.arg2] = cp->ports[r]->portval;
            break;
          }
      }
    }
    break;
  case INCRGLBL: /* val, target-ptr */
    if (IS_ACTIVE && (r = cp->reg[simdinstr.arg2],VALIDADDR(r))) {
      if (cp->col->colowner != cp->self ||
          row[ROW(r)].rowowner != cp->self) {
        if (ph==1) {
          printid(cp);
          printf(" incrglbl: not owner\n"); }
      } else {
        if (ph==2) {
          /* no cell arbitration is necessary */
          CELLAT(r).refc += simdinstr.arg1;
        }
      }
    }
    break;
  case CLRTRYING:
    if (ph==2 && IS_ACTIVE) {
      CLRBIT(cp->state,STATE_TRYING);
    }
    break;
  case DELETE:
    if (ph==2 && IS_ACTIVE) {
      cp->marks = 0;
      cp->refc = 0;
      cp->state = 0;
      cp->aux = 0;
      for (r=0; r<REGCNT; r++) cp->reg[r] = 0;
    }
    break;
  case COMMITMARK: /* mark */
    if (ph==2 && IS_ACTIVE) {
      if (TSTBITN(cp->marks,simdinstr.arg1)) {
        for (r=0; r<REGNUM; r++) {
          val = cp->reg[r];
          cp->reg[r] = cp->reg[r+REGNUM]; /*@@*/
          cp->reg[r+REGNUM] = val;
        }
      }
      SETBIT(cp->state,STATE_COMMIT);
    }
    break;
  case SWAP:
    if (ph==2 && IS_ACTIVE) {
      for (r=0; r<REGNUM; r++) {
        val = cp->reg[r];
        cp->reg[r] = cp->reg[r+REGNUM]; /*@@*/
        cp->reg[r+REGNUM] = val;
      }
    }
    break;
  case TSTINUSE:
    if (ph==2 && IS_ACTIVE && cp->refc <= 0)
      DEACTIVATE;
    break;
  case SETCOMMIT:
    if (ph==2 && IS_ACTIVE) {
      SETBIT(cp->state,STATE_COMMIT);
    }
    break;
  case TSTZERO: /* reg */
    if (ph==2 && IS_ACTIVE && !((cp->reg[simdinstr.arg1])==0))
      DEACTIVATE;
    break;
  case TSTNZERO: /* reg */
    if (ph==2 && IS_ACTIVE && ((cp->reg[simdinstr.arg1])==0))
      DEACTIVATE;
    break;
  case SELCELL: /* ptr */
    if (IS_ACTIVE && ph==2) {
      if (r = cp->reg[simdinstr.arg1],VALIDADDR(r)) {
        if (cp->col->colowner != cp->self ||
            row[ROW(r)].rowowner != cp->self) {
          printid(cp);
          printf(" selcell: not owner\n");
          printcell(cp);
          printaddr(cp->col->colowner); printf("\n");
          printaddr(row[ROW(r)].rowowner); printf("\n");
          printcelladdr(row[ROW(r)].rowowner);
        } else {
          CELLAT(r).aux = cp->self;
        }
      } else DEACTIVATE;
    }
    break;
  case ARBTILE:
    if (IS_ACTIVE) {
      if (ph==1) {
        SETBITN(cp->ports[NORTH]->portval,cp->portno);
      } else { /* ph ==2 */
        /* lower numbered bits have higher priority */
        if ((cp->ports[NORTH]->portval) & LOWERBITS(cp->portno)) DEACTIVATE;
      }
    }
    break;
  case SETRANDOM: /* reg */
    if (ph==2 && IS_ACTIVE) {
```

```
      cp->reg[simdinstr.arg1] = random();
    }
    break;
  case LOGXOR: /* treg sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] ^= cp->reg[simdinstr.arg2];
    break;
  case ROTATEL: /* treg sreg */
    if (ph==2 && IS_ACTIVE) {
      /* This is specifically a 16 bit operation and doesn't use
         a condition flag */
      val = cp->reg[simdinstr.arg2];
      val = ((val << 1) & 0xffff) + (val & 0x8000 ? 1 : 0);
      cp->reg[simdinstr.arg1] = val;
      }
    break;
  case ROTATER: /* treg sreg */
    if (ph==2 && IS_ACTIVE) {
      /* This is specifically a 16 bit operation and doesn't use
         a condition flag */
      val = cp->reg[simdinstr.arg2];
      val = ((val >> 1) & 0x7fff) + (val & 1 ? 0x8000 : 0);
      cp->reg[simdinstr.arg1] = val;
      }
    break;
  case SHIFTL: /* treg sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] = (cp->reg[simdinstr.arg2]) << 1;
    break;
  case SHIFTR: /* treg sreg */
    if (ph==2 && IS_ACTIVE)
      cp->reg[simdinstr.arg1] = ((unsigned int)(cp->reg[simdinstr.arg2])) >> 1;
    break;
  case SHIFTRA: /* treg sreg */
    if (ph==2 && IS_ACTIVE)
      /* This is probably not very portable, but produdces the
         desired effect on a SPARC */
      cp->reg[simdinstr.arg1] = (cp->reg[simdinstr.arg2]) >> 1;
    break;
  default:
    printf("Illegal op code %d\n",op);
    fflush(stdout);
    exit(-1);
  }
}
```

```
/*
-----------------------------------------------------------------
SIMD routines for Peano Fibonacci
-----------------------------------------------------------------
*/

#include <stdio.h>
#include "risc.h"
#include "basic.h"

/* ------------------------------------------------------------- */
#define ZERO 1
#define SUCC 2
#define FIBO 3
#define PLUS 4

/* ------------------------------------------------------------- */
/* Build the initial term */
term_build(n)
int n;
{
  int i;
  cellstate *cp,*sp;
  cp = alloc_in(ROWNUM-3,4);
  root = cp->self;
  sp = alloc_cell(cp);
  build_cell(cp,FIBO,2,sp,NULL);
  cp = sp;
  for (i=0; i<n; i++) {
    sp = alloc_cell(cp);
    build_cell(cp,SUCC,1,sp,NULL);
    cp = sp;
  }
  build_cell(cp,ZERO,1,NULL,NULL);
}

/* ------------------------------------------------------------- */
int reducedlim = 2;

/* ------------------------------------------------------------- */
simd_execute()
{
  int start;
  int flag,rcnt;

  progname = "fibo";
  do_init();
  fetchvers=0; storevers=fetchvers;
  stat_init();

/* ------------------------------------------------------------- */
  while (1) { /* beginning of main loop */
  stat_print_loop();

  flag = 0; /* Keep track of whether there are any fibo or plus tokens */

  /* rule (eq (fibo 0) 0) */
  do_posn(0); do_posnl(2,"fibo(0)=0");
  simd(INIT); SHOW;
  simd(ERASE); SHOW;
  simd1(CONST,ZERO); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,1); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,FIBO); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
resetglob();
simd(SENDGLOBAL); SHOW;
  simd1(SETMARK,2); SHOW;
if (testglob()) {
  flag = 1;
  simd2(TSTMARKPTR,1,LEFT); SHOW;
  simd1(CLEAR,NLEFT); SHOW;
  simd1(CLEAR,NRIGHT); SHOW;
  simd1(CONST,ZERO); SHOW;
  simd2(MOVE,NTOK,ACC); SHOW;
  simd(COMMIT); SHOW;
  simd1(CLRMARK,2); SHOW;
  simd1(SETMARK,1); SHOW;
  simd1(SETMARK,0); SHOW;

  /* rule (eq (fibo (succ 0)) (succ 0)) */
  /* use forward ? */
  do_posnl(2,"fibo(s 0) = (s 0)");
  simd(INIT); SHOW;
  simd1(TSTMARK,2); SHOW;
  simd2(SETMARKPTR,3,LEFT); SHOW;
  simd1(SETMARK,4); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,3); SHOW;
  simd1(CONST,SUCC); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,5); SHOW;
  simd2(TSTMARKPTR,1,LEFT); SHOW;
  simd1(SETMARK,6); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,4); SHOW;
  simd2(TSTMARKPTR,6,LEFT); SHOW;
  simd2(SETMARK,15); SHOW; /* Success */
  simd_fetch(NLEFT,LEFT,LEFT); SHOW;
  simd(INIT);
  simd1(TSTMARK,15); SHOW;
  simd_incr_any(1,NLEFT); SHOW;
  simd(INIT);
  simd1(TSTMARK,15); SHOW;
  simd1(CLEAR,NRIGHT); SHOW;
  simd1(CONST,SUCC); SHOW;
  simd2(MOVE,NTOK,ACC); SHOW;
  simd(COMMIT); SHOW;
  simd1(CLRMARK,2); SHOW;
  simd1(SETMARK,0); SHOW;

  /* rule (eq (fibo (succ (succ x))) (+ (fibo x) (fibo (succ x)))) */
  do_posnl(2,"fibo(s s x) = fibo(x) + fibo(s x)");
  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;
  simd2(SETMARKPTR,7,LEFT); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,7); SHOW;
  simd1(CONST,SUCC); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,9); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;
  simd2(TSTMARKPTR,9,LEFT); SHOW;
  simd1(SETMARK,10); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,4); SHOW;
```

```
  simd2(TSTMARKPTR,10,LEFT); SHOW;
  simd1(CLEAR,NRIGHT); SHOW;
  simd_alloc(NLEFT); SHOW;
  simd1(SETMARK,11); SHOW;
  simd_store(NLEFT,TOK,TOK); SHOW;
  SV0 { simd(INIT); SHOW; }
  SV0 { simd1(TSTMARK,11); SHOW; }
  simd_alloc(NRIGHT); SHOW;
  simd2(SETMARK,12); SHOW;  /* Success */
  simd2(SETMARKPTR,13,LEFT); SHOW;
  simd_store(NRIGHT,TOK,TOK); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,13); SHOW;
  simd_fetch(NTOK,LEFT,LEFT); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,12); SHOW;
  simd_fetch(NTOK,LEFT,NTOK); SHOW;
  FV0 { simd(INIT); SHOW;  }
  FV0 { simd1(TSTMARK,12); SHOW; }
  simd_store(NLEFT,LEFT,NTOK); SHOW;
  SV0 { simd(INIT); SHOW; }
  SV0 { simd1(TSTMARK,12); SHOW; }
  simd_incr_any(1,NTOK); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,12); SHOW;
  simd_fetch(NTOK,LEFT,LEFT); SHOW;
  FV0 { simd(INIT); SHOW; }
  FV0 { simd1(TSTMARK,12); SHOW; }
  simd_store(NRIGHT,LEFT,NTOK); SHOW;
  SV0 { simd(INIT); SHOW; }
  SV0 { simd1(TSTMARK,12); SHOW; }
  simd_incr_any(1,NTOK); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,12); SHOW;
  simd1(CONST,PLUS); SHOW;
  simd2(MOVE,NTOK,ACC); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,11); SHOW;
  simd1(COMMITMARK,12); SHOW;

  do_posnl(2,"setup");
  simd(INIT); SHOW;
  simd1(TSTMARK,0); SHOW;
  simd1(CONST,1); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  cntrod += 4;
}

if (size<2*mainloop) {

  /*
  do_commitfin();
  do_delete(); /*@@*/

  /* rule (eq (+ 0 x) x) */
  do_posnl(2,"0 + x = x");
  simd(INIT); SHOW;
  simd(ERASE); SHOW;
  simd1(CONST,ZERO); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,1); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,PLUS); SHOW;
  simd2(EQ,ACC,TOK); SHOW;

  resetglob();
  simd(SENDGLOBAL); SHOW;
  simd1(SETMARK,2); SHOW;
  if (testglob()) {
    flag = 1;
    simd2(TSTMARKPTR,1,LEFT); SHOW;
    simd1(CONST,FORWARD);
    simd2(MOVE,NTOK,ACC);
    simd2(MOVE,NLEFT,RIGHT);
    simd1(CLEAR,RIGHT);
    simd(COMMIT); SHOW;
    simd1(CLRMARK,2); SHOW;

    /* rule (eq (+ (succ x) y) (succ (+ x y))) */
    do_posnl(2,"(s x) + y = s(x + y)");
    simd(INIT); SHOW;
    simd1(TSTMARK,2); SHOW;
    simd2(SETMARKPTR,6,LEFT); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,6); SHOW;
    simd1(CONST,SUCC); SHOW;
    simd2(EQ,ACC,TOK); SHOW;
    simd1(SETMARK,8); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,2); SHOW;
    simd2(TSTMARKPTR,8,LEFT); SHOW;
    simd1(CLEAR,NRIGHT); SHOW;
    simd_alloc(NLEFT); SHOW;
    simd1(SETMARK,9); SHOW; /* Success */
    simd_store(NLEFT,TOK,TOK); SHOW;
    SV0 { simd(INIT); SHOW; }
    SV0 { simd1(TSTMARK,9); SHOW; }
    simd_fetch(NTOK,LEFT,LEFT); SHOW;
    FV0 { simd(INIT); SHOW; }
    FV0 { simd1(TSTMARK,9); SHOW; }
    simd_incr_any(1,NTOK); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,9); SHOW;
    simd_store(NLEFT,LEFT,NTOK); SHOW;
    SV0 { simd(INIT); SHOW; }
    SV0 { simd1(TSTMARK,9); SHOW; }
    simd_store(NLEFT,RIGHT,RIGHT); SHOW;
    SV0 { simd(INIT); SHOW; }
    SV0 { simd1(TSTMARK,9); SHOW; }
    simd1(CONST,SUCC); SHOW;
    simd2(MOVE,NTOK,ACC); SHOW;
    simd1(CLEAR,RIGHT); SHOW;
    simd(COMMIT); SHOW;
  }

  /* rule "propagate reduced" */
  do_posnl(2,"reduced");
  start = clock;
  simd(INIT); SHOW;
  simd1(CONST,1); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW;
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK,0); SHOW;

  simd(INIT); SHOW;
  simd1(TSTNOTMARK,0); SHOW;
  simd1(CONST,SUCC); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  /* simd2(TSTMARKPTR,0,LEFT); SHOW; */
```

```
  resetglob();
  simd(SETTRYING); SHOW;
  simd1(1STLOCAL,LEFT); SHOW;
  if (testglob()) {
    simd(CLRTRYING); SHOW;
    simd2(TSTMARKPTR,0,LEFT); SHOW;
    simd1(SETMARK,0); SHOW;
    rcnt = 0;
    while (1) {
      resetglob();
      simd(INITTRYING); SHOW;
      cntrednum++;
      simd(ARBTILE); SHOW;
      if (!testglob()) break;
      simd(ARBCOL); SHOW;
      simd1(SELROW,LEFT); SHOW;
      simd1(ARBROW,LEFT); SHOW;
      simd1(SELCELL,LEFT); SHOW;
      simd(CLRTRYING); SHOW;
      simd2(TSTMARKGLBL,LEFT,0); SHOW;
      simd1(SETMARK,0); SHOW;
      rcnt++;
      if (flag && reducedlim<rcnt) break;
    }
  }


  simd(INIT); SHOW;
  simd1(TSTMARK,0); SHOW;
  simd1(CONST,1); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  cntred += clock-start;
}

  if (CELLAT(root).CELLFLAGS&1) break;

  do_posn(1); do_posn1(2,"special");

  /* excluding actions in autonomous processes from basic counts */
  do_stat_save();
  special = 1;

  do_commitfin();
  do_relocate();
  do_forward();
  do_delete();

  special = 0;
  do_stat_restore();


  mainloop++;
  } /* end of main loop */

  do_final();
}
```

```
/*
-----------------------------------------------------------------------
SIMD routines for Conway's Game of LIFE
-----------------------------------------------------------------------
*/

#include <stdio.h>
#include "risc.h"
#include "basic.h"

/* ---------------------------------------------------------------------- */
/* Build the initial term */
term_build(ignore)
int ignore;
{
  int i,j,n;
  cellstate *cp,*sp,*temp;
  n=0; /* pdl assumes we always allocate in cell 0 */
       /* later we loop to make 4 layers */
  for (i=0; i<ROWNUM; i++) {
    for (j=0; j<COLNUM; j++) {
      temp = alloc_cell_in(i,j,n);
      if ((i==0) && (j==0)) {root = temp->self;};
    }
  }
  /* treat edges as permanent dead zone (null pointers) */
  for (i=0; i<ROWNUM; i++) {
    for (j=0; j<COLNUM; j++) {
      build_full_cell(&CELLREF(i,j,n),0,((i==4)?7:0), /* alive[i,j,n], */
                      ((j==0) ? NULL : &CELLREF(i,j-1,n)),
                      ((i==ROWNUM-1) ? NULL : &CELLREF(i+1,j,n)),
                      ((j==COLNUM-1) ? NULL : &CELLREF(i,j+1,n)),
                      ((i==0) ? NULL : &CELLREF(i-1,j,n)));
    }
  }
}

/* ---------------------------------------------------------------------- */
simd_execute()
{
  int start;
  int flag,rcnt;

  progname = "life";
  do_init();
  fetchvers=0; storevers=fetchvers;
  stat_init();

  while (1) { /* begin of main loop */
  stat_print_loop();

  flag = 0; /* Keep track of whether there are any fibo or pluses */

  /* Everyone starts with 0 neighbors */
  do_posn(0); do_posnl(2,"Start LIFE Main Loop");
  simd(INIT); SHOW;
  simd(ERASE); SHOW;
  simd1(CONST,0); SHOW;
  simd2(MOVE,TOK,ACC); SHOW; /* token holds number of neighbors */
  simd2(MOVE,NTOK,ACC); SHOW;   /* clear NTOK for later use */
  simd1(CONST,1); SHOW;     /* ACC holds constant 1 for increments */
  simd1(TSTNZERO,FLAGS); SHOW;
  simd1(SETMARK,2); SHOW;  /* set mark 2 if alive */
  simd2(ADD,TOK,ACC); SHOW;/* increment TOK if alive */

  /* begin prop east */
  do_posn(1); do_posnl(2,"Prop E-W");
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,2,RIGHT); SHOW;
  simd2(ADD,TOK,ACC); SHOW;

  /* begin prop west */
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,2,NRIGHT); SHOW;
  simd2(ADD,TOK,ACC); SHOW;

  /* setup for prop N-S */
  simd(INIT); SHOW;
  simd2(MOVE,NTOK,TOK); SHOW;
  simd2(LOGAND,ACC,NTOK); SHOW; /* low (1) bit */
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK, 3); SHOW;
  simd(INIT);
  simd1(CONST,2); SHOW; /* ACC is 2 */
  simd2(LOGAND,ACC,NTOK); SHOW; /* high (2) bit */
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK, 4); SHOW;

  /* begin prop north-south */
  do_posn(2); do_posnl(2,"Prop N-S");
  simd(INIT); SHOW;
  simd1(CONST,2); SHOW; /* ACC is 2 */
  simd2(TSTMARKPTR,4,LEFT); SHOW;
  simd2(ADD,TOK,ACC); SHOW;
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,4,NLEFT); SHOW;
  simd2(ADD,TOK,ACC); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,1); /* ACC is 1 again */
  simd2(TSTMARKPTR,3,LEFT); SHOW;
  simd2(ADD,TOK,ACC); SHOW;
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,3,NLEFT); SHOW;
  simd2(ADD,TOK,ACC); SHOW;

  /* begin life/death */
  do_posn(3); do_posnl(2,"Start Life/Death");
  simd(INIT); SHOW;
  simd1(CONST,3); SHOW;
  simd2(EQ,TOK,ACC); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd(INIT); SHOW;
  simd2(LT,TOK,ACC); SHOW;
  simd1(CLEAR,FLAGS); SHOW; /* He's dead, Jim. Undercrowded. 1- neighbors*/
  simd(INIT); SHOW;
  simd1(CONST,5); SHOW;
  simd2(GT,TOK,ACC); SHOW;
  simd1(CLEAR,FLAGS); SHOW; /* He's dead, Jim. Overcrowded. 5+ neighbors*/
  /* end of real loop */

  if ((CELLAT(root).CELLFLAGS)&1) break; /* quit when root alive */
  mainloop++;
  } /* end of main loop */
  do_final();
}
```

```
/* cc -O -o suncirc2 suncirc2.c */

/*
15 MIPS machine
assault% suncirc2 400
time taken:      52.167 ms
*/

#include <sys/types.h>
#include <sys/times.h>

main(argc,argv)
int argc;
char **argv;
{
  struct tms tim1,tim2;
  int hz;
  int i,scale,lim;
  int arg;

  sscanf(argv[1],"%d",&arg);

  times(&tim1);

  scale=10; /* @@ */

  lim = 1000/scale;
  for (i=0; i<lim; i++) {
    run(arg);
  }

  times(&tim2);

  hz = tim2.tms_utime - tim1.tms_utime + tim2.tms_stime - tim1.tms_stime;

  printf("time taken: %10.3f ms\n",hz*scale/60.0);
}


unsigned char dat[500];
unsigned char nxt[500];

#define NAND(x,y) (~((x)&(y)))
#define IOR(x,y) ((x)|(y))
#define TSTBIT(x,n) ((x)&(1<<n))

run(n)
int n;
{
  register int i,size,cc,nn,x;
  int cl,cln;
  size = 50;
  for (; 0<n; n--) {
    nn = 0;
    cc = cl;
    if (!(TSTBIT(cc,3) && TSTBIT(cc,3))) nn |= 1<<0;
    if ((TSTBIT(cc,0) || TSTBIT(cc,0))) nn |= 1<<1;
    if ((TSTBIT(cc,1) || TSTBIT(cc,1))) nn |= 1<<2;
    if ((TSTBIT(cc,2) || TSTBIT(cc,2))) nn |= 1<<3;
    cln = nn;
    x = cc&1;
    for (i=0; i<size; i++) {
      nn = 0;
      cc = dat[i];
      if (!(x&&x)) nn |= 1<<0;
      if (!(TSTBIT(cc,0)&&TSTBIT(cc,0))) nn |= 1<<1;
      if (!(TSTBIT(cc,1)&&TSTBIT(cc,1))) nn |= 1<<2;
      if (!(TSTBIT(cc,2)&&TSTBIT(cc,2))) nn |= 1<<3;
      if (!(TSTBIT(cc,3)&&TSTBIT(cc,3))) nn |= 1<<4;
      if (!(TSTBIT(cc,4)&&TSTBIT(cc,4))) nn |= 1<<5;
      if (!(TSTBIT(cc,5)&&TSTBIT(cc,5))) nn |= 1<<6;
      if (!(TSTBIT(cc,6)&&TSTBIT(cc,6))) nn |= 1<<7;
      x =  TSTBIT(cc,7);
      nxt[i] = nn;
    }
    cl = cln;
    for (i=0; i<size; i++) dat[i] = nxt[i];
/*
    printf("%1x",cl);
    for (i=0;i<35;i++) printf("%02x",dat[i]);
    putchar('\n');
*/
  }
}
```

```
/*
-----------------------------------------------------------------------
SIMD routines for 2D sort
-----------------------------------------------------------------------
*/

/* #define VERSION1 /* */
#define BASIC /* */

#include <stdio.h>
#include "risc.h"
#include "basic.h"

#include <stdlib.h>

#define NULLPTR 0
#define MARKN(x) (1<<(x))

/* ------------------------------------------------------------------- */
/* tokens */
#define SORTER 1

/* marks */
/* not used */
#define REDUCEDMARK 0
/* G0 and G1 constitute total order chain */
#define NUMBERMARK 1
#define G0 2
#define G1 3
#define G2 4
#define G3 5
#define M1 6
#define M2 7

/* ------------------------------------------------------------------- */
/* Mapping of abstract grid onto ensemble */

map2e(r,c,nr,nc,nn)
int r,c, *nr,*nc,*nn;
{
  int rn;
  rn = 0;
  if (r<ROWNUM) { *nr = r; } else { *nr = 2*ROWNUM-2 - r; rn += 2; }
  if (c<COLNUM) { *nc = c; } else { *nc = 2*COLNUM-2 - c; rn += 1; }
  *nn = rn;
}

/* produces an "address" */
int
map2a(r,c)
int r,c;
{
  int nr,nc,nn;
  nn = 0;
  if (r<ROWNUM) { nr = r; } else { nr = 2*ROWNUM-2 - r; nn += 2; }
  if (c<COLNUM) { nc = c; } else { nc = 2*COLNUM-2 - c; nn += 1; }
  /*
    {int res; res = ADDR(nr,nc,nn);
    printf("[%dx%d -> %d=%dx%d:%d]\n",r,c,res,ROW(res),COL(res),NUM(res)); }
  */
  return ADDR(nr,nc,nn);
}

/* ------------------------------------------------------------------- */

/* Build the initial term */
term_build(num)
int num;
{
  int i,j,r,c,n,ms,lp,rp,v,rows,cols,rnd;
  cellstate *cp;
  char *val;

  val = getenv("SEED");
  if (val != NULL) { rnd = 1; sscanf(val,"%d",&randomstate); }
  else rnd = 0;

  rows = (num+1)/2;
  cols = rows;
  if ((2*ROWNUM-1)<num || (2*COLNUM-1)<num) {
    printf("Size is too large: %d\n",num);
    exit(-1);
  }
  v = 1;
  for (j=0; j<2*cols-1; j++) {
    for (i=0; i<2*rows-1; i++) {
      map2e(i,j,&r,&c,&n);
      cp = alloc_cell_in(r,c,n);

      lp = NULLPTR;
      rp = NULLPTR;
      if (j&1) {
        if (i < (2*rows-2)) {
          lp = map2a(i+1,j);
          rp = map2a(i,j-1);
        } else {
          lp = map2a(i,j-1);
          /* NEXT OPTIONAL */
          rp = map2a(i,j-1);
        }
      } else {
        if (0<i) {
          lp = map2a(i-1,j);
          if (0<j) rp = map2a(i,j-1);
        } else {
          if (0<j) lp = map2a(i,j-1);
          /* NEXT OPTIONAL */
          if (0<j) rp = map2a(i,j-1);
        }
      }

      ms = MARKN(NUMBERMARK);
      if ((i+j)&1) { ms |= MARKN(G0); } else { ms |= MARKN(G1); }
      if (j&1) { ms |= MARKN(G2); } else { ms |= MARKN(G3); }
      if (j == 2*cols-2) ms |= MARKN(G2);

      cp->CELLTOK = SORTER;
      if (rnd)
        cp->CELLFLAGS = randoom();
      else
        cp->CELLFLAGS =
          ((i&1) ?
           (2000 - 32*i - j) :
           (2000 + 32*i - j)); /* alternate bad case 2 */
        /* cp->CELLFLAGS = 10000 - 32*i - j; /* bad case */
        /* cp->CELLFLAGS = 10000 + 32*i - j; /* alternate bad case */
        /* cp->CELLFLAGS = (2*cols-1)*(2*rows-2 - i) + (2*cols-2) - j; /* */
      v++;
      cp->marks = ms;
```

```
      cp->CELLLEFT = lp;
      cp->CELLRIGHT = rp;
    }
  }

  root = map2a(2*rows-2,2*cols-2);
  cp = &CELLAT(root);
  /* top node must assume priority */
  cp->marks |= MARKN(G0)|MARKN(G2);
}

showdat()
{
  int p,cnt;
  cellstate *cp;

  p = root;
  cnt = 0;
  while (p !=0) {
    cp = &CELLAT(p);
    p = cp->CELLLEFT;
    printf(" %d",cp->CELLFLAGS);
    cnt++;
    if (15<cnt) { printf("\n"); cnt=0; }
  }
  if (0<cnt) printf("\n");
}

/* ------------------------------------------------------------------ */
simd_execute()
{
  int start,s,flag,pntr,none0;

  progname = "2D sort";
  do_init();
  fetchvers=1; storevers=fetchvers; /* @@ */
  shownext = 0;
  stat_init();

  if (verbose) showdat();

  s = 1;
/* ------------------------------------------------------------------ */
/* 2D Sort */
/* ------------------------------------------------------------------ */
  while (1) { /* begin of main loop */

#ifdef BASIC
  s++; if (2<=s) s = 0;
  switch (s) {
  case 0:
    flag = G0;
    pntr = LEFT;
    none0 = 0;
    break;
  case 1:
    flag = G2;
    pntr = RIGHT;
    break;
  }
#else
  /* Improved worst case */
  s++; if (9<=s) s = 0;
  switch (s) {
  case 0:
  case 1:
  case 2:
  case 3:
  case 4:
    flag = G0;
    pntr = LEFT;
    none0 = 0;
    break;

  case 5:
  case 6:
  case 7:
  case 8:
    flag = G2;
    pntr = RIGHT;
    break;
  }
#endif

  stat_print_loop();


  do_posn(0);

  do_posnl(2,"sort start");
  simd(INIT); SHOW;
  simd1(TSTMARK,NUMBERMARK); SHOW;
  simd2(TSTMARKPTR,NUMBERMARK,pntr); SHOW;
  simd1(SETMARK,M1); SHOW;
  simd_fetch(NFLAGS,pntr,FLAGS); SHOW;

  do_posnl(2,"sort decide swap");
  simd(INIT); SHOW;
  simd1(TSTMARK,M1); SHOW;
  simd1(CLRMARK,M1); SHOW;
  simd2(GT,FLAGS,NFLAGS); SHOW;
  resetglob();
  simd(SENDGLOBAL); SHOW; cntred++;
  simd1(SETMARK,M1); SHOW;
  if (!testglob() && s == 0) break;

  do_posnl(2,"check overlapping");
  simd(INIT); SHOW;
  simd1(TSTMARK,M1); SHOW;
  simd1(TSTMARK,flag); SHOW;
  simd2(CLRMARKPTR,M1,pntr); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,M1); SHOW;
#ifdef VERSION1
  simd2(CLRMARKPTR,M1,pntr); SHOW;
#else
  simd2(TSTMARKPTR,M1,pntr); SHOW;
  simd2(CLRMARK,M1); SHOW;
#endif

  do_posnl(2,"sort perform swap");
  simd(INIT); SHOW;
  simd1(TSTMARK,M1); SHOW;
  simd1(CLRMARK,M1); SHOW;
  simd2(MOVE,ACC,FLAGS); SHOW;
  simd2(MOVE,FLAGS,NFLAGS); SHOW;
  simd_store(pntr,FLAGS,ACC); SHOW;
```

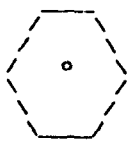
```
  mainloop++;
  } /* end of main loop */

  if (verbose) showdat();
  verbose = 0;
  do_final();
}
```

```
/*
-----------------------------------------------------------------------
SIMD routines for Circuit Simulation
-----------------------------------------------------------------------
*/

#include <stdio.h>
#include "risc.h"
#include "basic.h"

/* ------------------------------------------------------------------- */
/* tokens and marks */
#define NANDTOK 1
#define NANDMARK 4
#define ORTOK 2
#define ORMARK 5

/* ------------------------------------------------------------------- */
/* Build the initial term */

#define AT(x,y,i) (&CELLREF(x,y,i))

cellstate *
alloccell(x,y,i)
int x,y,i;
{
  cellstate *res;
  res = AT(x,y,i);
  if (TSTBIT(res->state,STATE_VALID))
    printf("cell %dx%d:%d already allocated\n",x,y,i);
  SETBIT(res->state,STATE_VALID);
  res->refc = 1;
  return res;
}

cellstate *
make_cell(x,y,i,tok,marks,flags,left,right)
cellstate *left,*right;
int x,y,i,tok,marks,flags;
{
  cellstate *cp;
  cp = alloccell(x,y,i);
  cp->CELLTOK = tok;
  cp->marks = marks;
  cp->CELLFLAGS = flags;
  if (left==NULL) cp->CELLLEFT = 0; else cp->CELLLEFT = left->self;
  if (right==NULL) cp->CELLRIGHT = 0; else cp->CELLRIGHT = right->self;
  return cp;
}

setptr(cp,r,ptr)
cellstate *cp,*ptr;
int r;
{
  if (ptr==NULL) cp->reg[r] = 0; else cp->reg[r] = ptr->self;
}

setreg(x,y,i,r,ptr)
cellstate *ptr;
int x,y,i,r;
{
  setptr(AT(x,y,i),r,ptr);
}
```

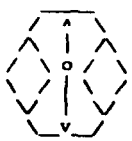
```
term_build(n)
int n;
{
  int i;
  cellstate *cp,*sp;
  int st,ndir,r,c;
  sp = make_cell(0,0,0,NANDTOK,0,0,NULL,NULL);
  root = sp->self;
  cp = make_cell(1,0,0,ORTOK,0,0,sp,sp);
  cp = make_cell(1,1,1,ORTOK,0,0,cp,cp);
  cp = make_cell(0,1,1,ORTOK,0,0,cp,cp);
  setptr(sp,LEFT,cp); setptr(sp,RIGHT,cp);
  for (i=0; i<500; i++) {
    cp = alloc_cell(sp);
    build_cell(cp,NANDTOK,0,sp,sp);
    sp = cp;
  }
}
```

```
/* ------------------------------------------------------------------ */
simd_execute()
{
  int start;

  progname = "circ";
  do_init();
  fetchvers=1; storevers=fetchvers;
  shownext = 0;
  stat_init();

  simd(INIT); SHOW;
  simd1(CONST,NANDTOK); SHOW;
  simd2(EQ,TOK,ACC); SHOW;
  simd1(SETMARK,NANDMARK); SHOW;

  simd(INIT); SHOW;
  simd1(CONST,ORTOK); SHOW;
  simd2(EQ,TOK,ACC); SHOW;
  simd1(SETMARK,ORMARK); SHOW;

  termshow = 0;

  while (1) { /* begin of main loop */
  stat_print_loop();

/* ------------------------------------------------------------------ */
/* Bubble Sort */

/* ------------------------------------------------------------------ */
  do_posn(0);
  do_posn1(2,"nand");

  simd(INIT); SHOW;
  simd1(TSTMARK,NANDMARK); SHOW;
  simd(SETTRYING); SHOW;
  simd2(TSTMARKPTR,0,LEFT); SHOW;
  simd2(TSTMARKPTR,0,RIGHT); SHOW;
  simd1(CLRMARK,0); SHOW;
  simd(INITTRYING); SHOW;
  simd2(TSTNOTMARKPTR,0,LEFT); SHOW;
  simd1(SETMARK,0); SHOW;
  simd(INITTRYING); SHOW;
  simd2(TSTNOTMARKPTR,0,RIGHT); SHOW;
  simd1(SETMARK,0); SHOW;

  do_posn1(2,"or");

  simd(INIT); SHOW;
  simd1(TSTMARK,ORMARK); SHOW;
  simd(SETTRYING); SHOW;
  simd2(TSTNOTMARKPTR,0,LEFT); SHOW;
  simd2(TSTNOTMARKPTR,0,RIGHT); SHOW;
  simd1(CLRMARK,0); SHOW;
  simd(INITTRYING); SHOW;
  simd2(TSTMARKPTR,0,LEFT); SHOW;
  simd1(SETMARK,0); SHOW;
  simd(INITTRYING); SHOW;
  simd2(TSTMARKPTR,0,RIGHT); SHOW;
  simd1(SETMARK,0); SHOW;

  do_posn(1);

  mainloop++;
  if (size <= mainloop) break;

  } /* end of main loop */

  verbose = 0;

  do_final();
}
```

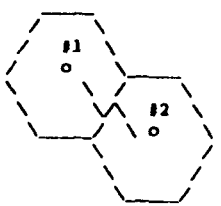
```
/*************************************************************************
SIMD routines for Finite Element Method --- Hex Grid

Each RRM Cell contains one hex grid element, which has 6 particle
slots, one in each of 6 directions.  These bits are kept in the
lowest 6 bits of the FLAGS register.

We keep pointers to NESW neighbors in R,L,NR,NL registers.
The extra two neighbors (NE and SW) are tested through the
N and S nieghbors, and the information passed on.

The model is a hexagonal grid of space.  Between each
neighboring grid element, there is space for two particles,
one in each direction.  A particle continues in a straight
line unless it collides with some other particle at the
center of a hex grid.  Otherwise it passes through.
All particles have unit mass, unit velocity.  In one timestep
(one big loop) particles exit one cell, enter another, and
possibly collide.  The collision rules are the hardest to encode.
All collisions preserve the total momentum and the number of particles.

The model is implemented by having one RRM cell represent each
hexagonal grid element.  At the top of the big loop, each cell's
FLAGS register contains 6 bits determining the presence or
absence of 6 particles exiting the cell in each of 6 directions.
A grid element can be thought of as follows:
```



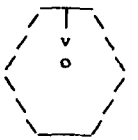
```
the "o" is the center of the element.
```



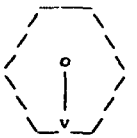
```
Here, the ^,/,\,v are supposed to be arrows.
The cell contains 6 possible particles, at most one
on each of the above arrows.  These bits (yes or no,
a particle is on that track) are represented by the
lowermost 6 bits of the the FLAGS register.
```




```
The movement proceeds in two phases.
In the first phase, any particle in element #1 on the
indicated path is passed on to element #2.  (the particle
moves in its indicated direction).  In the second phase,
particles heading inward toward the center of an element
move through the center, and arrive on the path leading
out from the grid element.
```

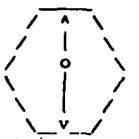


becomes



```
In most cases, the particles move unimpeded through the center
point of a grid element.  However, if two particles meet head-on,
they will deflect:
```

becomes



```
That is, two particles collide, and scatter through 60 degrees.
The same may happen if three or four particles collide.
In this model, collisions preserve momentum and number of
particles, which nearly determines all the collision rules.
The only degree of freedom remaining is the choice of scattering
head-on collisions 60 degrees clockwise, or 60 degrees
counter-clockwise.
*************************************************************************/

#include <stdio.h>
#include "risc.h"
#include "basic.h"

#define ALLDIRS 63

/* ------------------------------------------------------------------------ */
/* Build the initial term */
/* Similar to SIMD-LIFE  */
term_build(ignore)
int ignore;
{
  int i,j,n;
  cellstate *cp,*sp,*temp;
  n=0; /* pdl assumes we always allocate in cell 0 */
       /* later we loop to make 4 layers? (flip) */
```

```
  for (i=0; i<ROWNUM; i++) {
    for (j=0; j<COLNUM; j++) {
      temp = alloc_cell_in(i,j,n);
      if ((i==0) && (j==0)) {root = temp->self;};
    }
  }
  /* treat edges as permanent dead zone (null pointers) */
  for (i=0; i<ROWNUM; i++) {
    for (j=0; j<COLNUM; j++) {
      build_full_cell(&CELLREF(i,j,n),0,((((i*j)/5) & ALLDIRS),/*state[i,j,n]*/
                      /* note that ((i*j)/5) gives a distribution */
                      /* with no particles in the 0,0 area, and lots */
                      /* of particles in the COLNUM,ROWNUM area */
                      /* so we can detect termination by presence */
                      /* of particles in cell 0,0 */
                      ((j==0) ? NULL : &CELLREF(i,j-1,n)),
                      ((i==ROWNUM-1) ? NULL : &CELLREF(i+1,j,n)),
                      ((j==COLNUM-1) ? NULL : &CELLREF(i,j+1,n)),
                      ((i==0) ? NULL : &CELLREF(i-1,j,n)));
    }
  }
}

/* ------------------------------------------------------------------- */
simd_execute()
{
  progname = "fluid";
  do_init();
  fetchvers=0; storevers=fetchvers;
  stat_init();

  while (1) { /* begin of main loop */
  stat_print_loop();

  /* Start */
  /* FLAGS now has good bits */
  do_posn(0); do_posnl(2,"Start Fluid Main Loop");
  /* transfer FLAGS bits into marks 1-6 */
  simd(INIT); SHOW;
  simd(ERASE); SHOW;
  simd1(CONST,1); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW; /* is the 1st bit ON? */
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK,1); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,2); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW; /* is the 2nd bit ON? */
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK,2); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,4); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW; /* is the 3rd bit ON? */
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK,3); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,8); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW; /* is the 4th bit ON? */
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK,4); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,16); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW; /* is the 5th bit ON? */
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK,5); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,32); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW; /* is the 6th bit ON? */
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK,6); SHOW;
  /* begin diagonal propagation */
  do_posn(1); do_posnl(2,"Prop Diagonal");
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,5,LEFT); SHOW;
  simd1(SETMARK,7); SHOW;
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,2,NRIGHT); SHOW;
  simd1(SETMARK,8); SHOW;
  /* begin real propagation */
  do_posn(2); do_posnl(2,"Propagate All");
  simd(INIT); SHOW;
  simd1(CONST,0); SHOW;
  simd2(MOVE,FLAGS,ACC); /* flags = 0 */
  simd2(TSTMARKPTR,4,RIGHT); SHOW;
  simd1(CONST,8); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,7,RIGHT); SHOW;
  simd1(CONST,16); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,6,LEFT); SHOW;
  simd1(CONST,32); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,1,NRIGHT); SHOW;
  simd1(CONST,1); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,8,NRIGHT); SHOW;
  simd1(CONST,2); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  simd(INIT); SHOW;
  simd2(TSTMARKPTR,3,NLEFT); SHOW;
  simd1(CONST,4); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  do_posn(4); do_posnl(2,"Do Collisions");
  /* first three */
  simd(INIT); SHOW;
  simd(CLRMARK,15); /* clear this for done-test */
  simd1(CONST,21); SHOW;
  simd2(EQ,ACC,FLAGS); SHOW;
  simd1(CONST,42); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(SETMARK,15);  SHOW;
  /* second three */
  simd(INIT); SHOW;
  simd1(CONST,42); SHOW;
  simd2(EQ,ACC,FLAGS); SHOW;
  simd1(TSTNOTMARK,15); SHOW; /* didn't just become this */
  simd1(CONST,21); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  /* first four */
  simd(INIT); SHOW;
  simd1(CONST,54); SHOW;
  simd2(EQ,ACC,FLAGS); SHOW;
  simd1(CONST,27); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(SETMARK,15);  SHOW;
```

```
  /* second four */
  simd(INIT); SHOW;
  simd1(CONST,45); SHOW;
  simd2(EQ,ACC,FLAGS); SHOW; /* note no need for 15 check */
  simd1(CONST,54); SHOW;      /* since 54 not created */
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(SETMARK,15);  SHOW;
  /* third four */
  simd(INIT); SHOW;
  simd1(CONST,27); SHOW;
  simd2(EQ,ACC,FLAGS); SHOW;
  simd1(TSTNOTMARK,15); SHOW; /* didn't just become this */
  simd1(CONST,45); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(SETMARK,15);  SHOW;
/* start opposition ONE */
  do_posn(5); do_posnl(2,"Do Collisions (opp1)");
  simd(INIT); SHOW;
  simd1(CONST,9); SHOW;
  simd2(MOVE,NFLAGS,ACC); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(SETMARK,16); SHOW; /* 16 is important case */
  simd1(SETMARK,15); SHOW; /* 15 we are done now */
  simd1(CONST,54); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW;
  simd2(MOVE,NFLAGS,ACC); SHOW;
  /* first three */
  simd1(CONST,2); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,38); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16);  SHOW;
  /* second three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,4); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,22); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* third three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,16); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,52); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* fourth three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,32); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,50); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* first two */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,0); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,18); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
/* start opposition TWO */
  do_posn(6); do_posnl(2,"Do Collisions (opp 2)");
  simd(INIT); SHOW;
  simd1(CONST,18); SHOW;
  simd2(MOVE,NFLAGS,ACC); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(TSTNOTMARK,15); SHOW; /* didn't just become this */
  simd1(SETMARK,16); SHOW; /* 16 is important case */
  simd1(SETMARK,15); SHOW; /* 15 we are done now */
  simd1(CONST,45); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW;
  simd2(MOVE,NFLAGS,ACC); SHOW;
  /* first three */
  simd1(CONST,4); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,13); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* second three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,8); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,44); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* third three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,32); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,41); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* fourth three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,1); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,37); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* first two */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,0); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,36); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
/* start opposition THREE */
  do_posn(7); do_posnl(2,"Do Collisions (opp 3)");
  simd(INIT); SHOW;
  simd1(CONST,36); SHOW;
  simd2(MOVE,NFLAGS,ACC); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(TSTNOTMARK,15); SHOW; /* didn't just become this */
  simd1(SETMARK,16); SHOW; /* 16 is important case */
  simd1(SETMARK,15); SHOW; /* 15 we are done now */
  simd1(CONST,27); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW;
```

```
  simd2(MOVE,NFLAGS,ACC); SHOW;
  /* first three */
  simd1(CONST,8); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,26); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* second three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,16); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,25); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* third three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,1); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,19); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* fourth three */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,2); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,11); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;
  simd1(CLRMARK,16); SHOW;
  /* first two */
  simd(INIT); SHOW;
  simd1(TSTMARK,16); SHOW;
  simd1(CONST,0); SHOW;
  simd2(EQ,ACC,NFLAGS); SHOW;
  simd1(CONST,9); SHOW;
  simd2(MOVE,FLAGS,ACC); SHOW;

  /* end oppositions */
  do_posn(8); do_posn1(2,"End of collisions");
  /* now FLAGS is ready for next iteration */
  /* end of real loop */

  if ((CELLAT(root).CELLFLAGS)&1) break; /* quit when particle there */
  mainloop++;
  } /* end of main loop */
  do_final();
}
```

```
/*
-----------------------------------------------------------------------
SIMD routines for Numeric Fibonacci
-----------------------------------------------------------------------
*/

#include <stdio.h>
#include "risc.h"
#include "basic.h"

/* ------------------------------------------------------------------- */
#define NUMB 1
#define FIBO 2
#define PLUS 3
#define FIBOHOLD 4

/* ------------------------------------------------------------------- */
/* Build the initial term */
term_build(n)
int n;
{
  int i;
  cellstate *cp,*sp;
  cp = alloc_in(ROWNUM-3,4);
  root = cp->self;
  sp = alloc_cell(cp);
  build_cell(cp,FIBO,0,sp,NULL);
  cp = sp;
  build_cell(cp,NUMB,n,NULL,NULL);
}

/* ------------------------------------------------------------------- */
int reducedlim = 2;

/* ------------------------------------------------------------------- */
simd_execute()
{
  int start;
  int flag;
  int opt = 1;

  opt = 1;

  progname = "nfib";
  do_init();
  fetchvers=0; storevers=fetchvers;
  stat_init();

  while (1) { /* begin of main loop */
  stat_print_loop();

  flag = 0; /* Keep track of whether there are any fibo or pluses */

/* ------------------------------------------------------------------- */
  do_posn(0);

  /* fibo */
  do_posn1(2,"fibo base");
  simd(INIT); SHOW;
  simd(ERASE); SHOW;
  simd1(CONST,NUMB); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,1); SHOW;
    if (CELLAT(root).marks&(1<<1)) break;

    simd(INIT); SHOW;
    simd1(CONST,FIBO); SHOW;
    simd2(EQ,ACC,TOK); SHOW;
  resetglob();
  simd(SENDGLOBAL); SHOW;
    simd1(SETMARK,2); SHOW;
  if (testglob()) {
    flag = 1;
    simd2(TSTMARKPTR,1,LEFT); SHOW;
    simd1(SETMARK,3); SHOW;
    simd_fetch(NFLAGS,LEFT,FLAGS); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,3); SHOW;
    simd1(CONST,1); SHOW;
    simd2(LE,NFLAGS,ACC); SHOW;
    simd_incrptr(-1,LEFT); SHOW;
    simd1(CONST,NUMB); SHOW;
    simd2(MOVE,TOK,ACC); SHOW;
    simd2(MOVE,FLAGS,NFLAGS); SHOW;
    simd1(CLRMARK,3); SHOW;
    if (opt) { simd1(CLEAR,LEFT); SHOW; }

    do_posn1(2,"fibo gen");
    simd(INIT); SHOW;
    simd1(TSTMARK,3); SHOW;
    simd_alloc(NLEFT); SHOW;
    simd1(SETMARK,4); SHOW;
    simd_alloc(NRIGHT); SHOW;
    simd1(SETMARK,5); SHOW;
    simd2(SETMARKPTR,6,NLEFT); SHOW;
    simd2(SETMARKPTR,7,NRIGHT); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,6); SHOW;
    simd_alloc(LEFT); SHOW;
    simd1(SETMARK,8); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,7); SHOW;
    simd_alloc(LEFT); SHOW;
    simd1(SETMARK,8); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,5); SHOW;
    simd2(TSTMARKPTR,8,NLEFT); SHOW;
    simd2(TSTMARKPTR,8,NRIGHT); SHOW;
    simd2(SETMARKPTR,6,NRIGHT); SHOW;
    simd1(SETMARK,9); SHOW;
    simd1(CONST,1); SHOW;
    simd2(SUB,NFLAGS,ACC);  SHOW;
    simd_store(NLEFT,NFLAGS,NFLAGS); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,9); SHOW;
    simd2(SUB,NFLAGS,ACC);  SHOW;
    simd_store(NRIGHT,NFLAGS,NFLAGS); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,6); SHOW;
    simd2(CONST,FIBO); SHOW;
    simd2(MOVE,TOK,ACC); SHOW;
    simd2(SETMARKPTR,10,LEFT); SHOW;
    simd_store(LEFT,FLAGS,NFLAGS); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,10); SHOW;
    simd1(CONST,NUMB); SHOW;
    simd2(MOVE,TOK,ACC); SHOW;
```

```
  simd(INIT); SHOW;
  simd1(TSTMARK,4); SHOW;
  simd1(CONST,PLUS); SHOW;
  simd2(MOVE,NTOK,ACC); SHOW;
  simd1(CLEAR,NFLAGS); SHOW;
  simd2(COMMITMARK,9); SHOW;

  do_posn1(2,"fibo hold");
  /* new */
  simd(INIT); SHOW;
  simd1(TSTMARK,7); SHOW;
  simd1(CONST,10); SHOW;
  simd2(LT,ACC,NFLAGS); SHOW;
  simd1(CONST,FIBOHOLD); SHOW;
  simd2(MOVE,TOK,ACC); SHOW;
}

/* ------------------------------------------------------------------ */
  /* plus */
  do_posn1(2,"plus");
  simd(INIT); SHOW;
  simd1(CONST,PLUS); SHOW;
  simd2(EQ,TOK,ACC); SHOW;
resetglob();
simd(SENDGLOBAL); SHOW;
  simd2(TSTMARKPTR,1,LEFT); SHOW;
if (testglob()) {
  flag = 1;
  simd2(TSTMARKPTR,1,RIGHT); SHOW;
  simd1(SETMARK,11); SHOW;
  simd_fetch(FLAGS,LEFT,FLAGS); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,11); SHOW;
  simd_fetch(ACC,RIGHT,FLAGS); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,11); SHOW;
  simd2(ADD,FLAGS,ACC); SHOW;
  simd_incrptr(-1,LEFT); SHOW;
  simd_incrptr(-1,RIGHT); SHOW;
  if (opt) { simd1(CLEAR,LEFT); SHOW; simd1(CLEAR,RIGHT); SHOW; }
  simd1(CONST,NUMB); SHOW;
  simd2(MOVE,TOK,ACC); SHOW;

  /* new */
  do_posn1(2,"fibo release");
  simd(INIT); SHOW;
  simd1(CONST,PLUS); SHOW;
  simd2(EQ,TOK,ACC); SHOW;
  simd2(TSTMARKPTR,1,LEFT); SHOW;
  simd2(SETMARKPTR,12,RIGHT); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,12); SHOW;
  simd1(CONST,FIBOHOLD); SHOW;
  simd2(EQ,TOK,ACC); SHOW;
  simd1(CONST,FIBO); SHOW;
  simd2(MOVE,TOK,ACC); SHOW;
}

  do_posn(1); do_posn1(2,"special");

  /* excluding actions in autonomous processes from basic counts */
  do_stat_save();
  special = 1;
```



```
  do_commitfin(); /* was last, then second */
  do_relocate();
  do_forward();
  do_delete();

  special = 0;
  do_stat_restore();

  mainloop++;
  } /* end of main loop */

  do_final();
}
```

simd-peano-fibo.c


```
/*
-----------------------------------------------------------------------
SIMD routines for Peano Fibonacci
-----------------------------------------------------------------------
*/

#include <stdio.h>
#include "risc.h"
#include "basic.h"

/* ---------------------------------------------------------------------- */
#define ZERO 1
#define SUCC 2
#define FIBO 3
#define PLUS 4

/* ---------------------------------------------------------------------- */
/* Build the initial term */
term_build(n)
int n;
{
  int i;
  cellstate *cp,*sp;
  cp = alloc_in(ROWNUM-3,4);
  root = cp->self;
  sp = alloc_cell(cp);
  build_cell(cp,FIBO,2,sp,NULL);
  cp = sp;
  for (i=0; i<n; i++) {
    sp = alloc_cell(cp);
    build_cell(cp,SUCC,1,sp,NULL);
    cp = sp;
  }
  build_cell(cp,ZERO,1,NULL,NULL);
}

/* ---------------------------------------------------------------------- */
int reducedlim = 2;

/* ---------------------------------------------------------------------- */
simd_execute()
{
  int start;
  int flag,rcnt;

  progname = "fibo";
  do_init();
  fetchvers=0; storevers=fetchvers;
  stat_init();

  while (1) { /* begin of main loop */
  stat_print_loop();

  flag = 0; /* Keep track of whether there are any fibo or pluses */

  /* rule (eq (fibo 0) 0) */
  do_posn(0); do_posnl(2,"fibo(0)=0");
  simd(INIT); SHOW;
  simd(ERASE); SHOW;
  simd1(CONST,ZERO); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,1); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,FIBO); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
resetglob();
simd(SENDGLOBAL); SHOW;
  simd1(SETMARK,2); SHOW;
if (testglob()) {
  flag = 1;
  simd2(TSTMARKPTR,1,LEFT); SHOW;
  simd1(CLEAR,NLEFT); SHOW;
  simd1(CLEAR,NRIGHT); SHOW;
  simd1(CONST,ZERO); SHOW;
  simd2(MOVE,NTOK,ACC); SHOW;
  simd(COMMIT); SHOW;
  simd1(CLRMARK,2); SHOW;
  simd1(SETMARK,1); SHOW;
  simd1(SETMARK,0); SHOW;

  /* rule (eq (fibo (succ 0)) (succ 0)) */
  /* use forward ? */
  do_posnl(2,"fibo(s 0) = (s 0)");
  simd(INIT); SHOW;
  simd1(TSTMARK,2); SHOW;
  simd2(SETMARKPTR,3,LEFT); SHOW;
  simd1(SETMARK,4); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,3); SHOW;
  simd1(CONST,SUCC); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,5); SHOW;
  simd2(TSTMARKPTR,1,LEFT); SHOW;
  simd1(SETMARK,6); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,4); SHOW;
  simd2(TSTMARKPTR,6,LEFT); SHOW;
  simd2(SETMARK,15); SHOW; /* Success */
  simd_fetch(NLEFT,LEFT,LEFT); SHOW;
  simd(INIT);
  simd1(TSTMARK,15); SHOW;
  simd_incr_any(1,NLEFT); SHOW;
  simd(INIT);
  simd1(TSTMARK,15); SHOW;
  simd1(CLEAR,NRIGHT); SHOW;
  simd1(CONST,SUCC); SHOW;
  simd2(MOVE,NTOK,ACC); SHOW;
  simd(COMMIT); SHOW;
  simd1(CLRMARK,2); SHOW;
  simd1(SETMARK,0); SHOW;

  /* rule (eq (fibo (succ (succ x))) (+ (fibo x) (fibo (succ x)))) */
  do_posnl(2,"fibo(s s x) = fibo(x) + fibo(s x)");
  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;
  simd2(SETMARKPTR,7,LEFT); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,7); SHOW;
  simd1(CONST,SUCC); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,9); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;
  simd2(TSTMARKPTR,9,LEFT); SHOW;
  simd1(SETMARK,10); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,4); SHOW;
  simd2(TSTMARKPTR,10,LEFT); SHOW;
```

```
  simd1(CLEAR,NRIGHT); SHOW;
  simd_alloc(NLEFT); SHOW;
  simd1(SETMARK,11); SHOW;
  simd_store(NLEFT,TOK,TOK); SHOW; /*vers*/
  SV0 [ simd(INIT); SHOW; ]
  SV0 [ simd1(TSTMARK,11); SHOW; ]
  simd_alloc(NRIGHT); SHOW;
  simd2(SETMARK,12); SHOW;  /* Success */
  simd2(SETMARKPTR,13,LEFT); SHOW;
  simd_store(NRIGHT,TOK,TOK); SHOW;  /* should always succeed */
  simd(INIT); SHOW;
  simd1(TSTMARK,13); SHOW;
  simd_fetch(NTOK,LEFT,LEFT); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,12); SHOW;
  simd_fetch(NTOK,LEFT,NTOK); SHOW;
  FV0 [ simd(INIT); SHOW;  ]
  FV0 [ simd1(TSTMARK,12); SHOW; ]
  simd_store(NLEFT,LEFT,NTOK); SHOW;
  SV0 [ simd(INIT); SHOW; ]
  SV0 [ simd1(TSTMARK,12); SHOW; ]
  simd_incr_any(1,NTOK); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,12); SHOW;
  simd_fetch(NTOK,LEFT,LEFT); SHOW;
  FV0 [ simd(INIT); SHOW; ]
  FV0 [ simd1(TSTMARK,12); SHOW; ]
  simd_store(NRIGHT,LEFT,NTOK); SHOW;
  SV0 [ simd(INIT); SHOW; ]
  SV0 [ simd1(TSTMARK,12); SHOW; ]
  simd_incr_any(1,NTOK); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,12); SHOW;
  simd1(CONST,PLUS); SHOW;
  simd2(MOVE,NTOK,ACC); SHOW;
  simd(INIT); SHOW;
  simd1(TSTMARK,11); SHOW;
  simd1(COMMITMARK,12); SHOW;

  /* new */
  do_posnl(2,"setup");
  simd(INIT); SHOW;
  simd1(TSTMARK,0); SHOW;
  simd1(CONST,1); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  cntred += 4;
}

if (size<2*mainloop) {

  /*
  do_commitfin();
  do_delete(); /*@@*/

  /* rule (eq (+ 0 x) x) */
  do_posnl(2,"0 + x = x");
  simd(INIT); SHOW;
  simd(ERASE); SHOW;
  simd1(CONST,ZERO); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,1); SHOW;
  simd(INIT); SHOW;
  simd1(CONST,PLUS); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  resetglob();
  simd(SENDGLOBAL); SHOW;
    simd1(SETMARK,2); SHOW;
  if (testglob()) {
    flag = 1;
    simd2(TSTMARKPTR,1,LEFT); SHOW;
  #if 0
    simd1(RELOCREQ,RIGHT); SHOW;
    simd1(SETMARK,4); SHOW;
    simd_fetch(NLEFT,RIGHT,LEFT); SHOW;
    FV0 [ simd(INIT); SHOW; ]
    FV0 [ simd1(TSTMARK,4); SHOW; ]
    simd_incr_any(1,NLEFT); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,4); SHOW;
    simd_fetch(NRIGHT,RIGHT,RIGHT); SHOW;
    FV0 [ simd(INIT); SHOW; ]
    FV0 [ simd1(TSTMARK,4); SHOW; ]
    simd_incr_any(1,NRIGHT); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,4); SHOW;
    simd_fetch(NFLAGS,RIGHT,FLAGS); SHOW;
    FV0 [ simd(INIT); SHOW; ]
    FV0 [ simd1(TSTMARK,4); SHOW; ]
    simd_fetch(NTOK,RIGHT,TOK); SHOW;
    FV0 [ simd(INIT); SHOW; ]
    FV0 [ simd1(TSTMARK,4); SHOW; ]
  #else
    simd1(CONST,FORWARD);
    simd2(MOVE,NTOK,ACC);
    simd2(MOVE,NLEFT,RIGHT);
    simd1(CLEAR,RIGHT);
  #endif
    simd(COMMIT); SHOW;
    simd1(CLRMARK,2); SHOW;

    /* rule (eq (+ (succ x) y) (succ (+ x y))) */
    do_posnl(2,"(s x) + y = s(x + y)");
    simd(INIT); SHOW;
    simd1(TSTMARK,2); SHOW;
    simd2(SETMARKPTR,6,LEFT); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,6); SHOW;
    simd1(CONST,SUCC); SHOW;
    simd2(EQ,ACC,TOK); SHOW;
    simd1(SETMARK,8); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,2); SHOW;
    simd2(TSTMARKPTR,8,LEFT); SHOW;
    simd1(CLEAR,NRIGHT); SHOW;
    simd_alloc(NLEFT); SHOW;
    simd1(SETMARK,9); SHOW; /* Success */
    simd_store(NLEFT,TOK,TOK); SHOW;
    SV0 [ simd(INIT); SHOW; ]
    SV0 [ simd1(TSTMARK,9); SHOW; ]
    simd_fetch(NTOK,LEFT,LEFT); SHOW; /* this and next omitted because ? */
    FV0 [ simd(INIT); SHOW; ]
    FV0 [ simd1(TSTMARK,9); SHOW; ]
    simd_incr_any(1,NTOK); SHOW;
    simd(INIT); SHOW;
    simd1(TSTMARK,9); SHOW;
    simd_store(NLEFT,LEFT,NTOK); SHOW;
    SV0 [ simd(INIT); SHOW; ]
    SV0 [ simd1(TSTMARK,9); SHOW; ]
```

```
  simd_store(NLEFT,RIGHT,RIGHT); SHOW;
  SVO [ simd(INIT); SHOW; ]
  SVO [ simd1(TSTMARK,9); SHOW; ]
  simd1(CONST,SUCC); SHOW;
  simd2(MOVE,NTOK,ACC); SHOW;
  simd1(CLEAR,RIGHT); SHOW;
  simd(COMMIT); SHOW;
]

  /* rule "propagate reduced" */
  do_posn1(2,"reduced");
  start = clock;
  simd(INIT); SHOW;
  simd1(CONST,1); SHOW;
  simd2(LOGAND,ACC,FLAGS); SHOW;
  simd1(TSTNZERO,ACC); SHOW;
  simd1(SETMARK,0); SHOW;

  simd(INIT); SHOW;
  simd1(TSTNOTMARK,0); SHOW;
  simd1(CONST,SUCC); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  /* simd2(TSTMARKPTR,0,LEFT); SHOW; */
  resetglob();
  simd(SETTRYING); SHOW;
  simd1(TSTLOCAL,LEFT); SHOW;
  if (testglob()) [
    simd(CLRTRYING); SHOW;
    simd2(TSTMARKPTR,0,LEFT); SHOW;
    simd1(SETMARK,0); SHOW;
    rcnt = 0;
    while (1) {
      resetglob();
      simd(INITTRYING); SHOW;
      cntrednum++;
      simd(ARBTILE); SHOW;
      if (!testglob()) break;
      simd(ARBCOL); SHOW;
      simd1(SELROW,LEFT); SHOW;
      simd1(ARBROW,LEFT); SHOW;
      simd1(SELCELL,LEFT); SHOW;
      simd(CLRTRYING); SHOW;
      simd2(TSTMARKGLBL,LEFT,0); SHOW;
      simd1(SETMARK,0); SHOW;
      rcnt++;
      if (flag && reducedlim<rcnt) break;
    ]
  ]


  simd(INIT); SHOW;
  simd1(TSTMARK,0); SHOW;
  simd1(CONST,1); SHOW;
  simd2(LOGIOR,FLAGS,ACC); SHOW;
  cntred += clock-start;
]

/*
  simd(INIT); SHOW;
  simd1(CONST,3); SHOW;
  simd2(EQ,FLAGS,ACC); SHOW;
  resetglob();
  simd(SENDGLOBAL); SHOW;
  simd(NOP); SHOW;
```

```
  if (testglob()) break;
*/
  if (CELLAT(root).CELLFLAGS&1) break;

  do_posn(1); do_posn1(2,"special");

  /* excluding actions in autonomous processes from basic counts */
  do_stat_save();
  special = 1;

  do_commitfin(); /* was last, then second */
  do_relocate();
  do_forward();
  do_delete();

  special = 0;
  do_stat_restore();


  mainloop++;
  } /* end of main loop */

  do_final();
}
```

```
/*
-----------------------------------------------------------------
SIMD routines for the Takeutchi function
-----------------------------------------------------------------
*/

#include <stdio.h>
#include "risc.h"
#include "basic.h"

do_commitfin_lrf()
{
  int start;
  start = clock;
  if (TSTDETAIL(10)) printf("===> do_commitfin_lrf start ============\n");
  simd(INIT);
  simd1(TSTSTATE,STATE_COMMIT);
  if (displaysubs) SHOW;
  resetglob();
  simd(SENDGLOBAL);
  simd1(TSTADDR,NLEFT); /* was NOP, now use simd_incr_addr below */
  if (testglob()) {
    simd_incr_addr(-1,NLEFT);
    simd(INIT);
    simd1(TSTSTATE,STATE_COMMIT);
    simd_incr_any(-1,NRIGHT);
    simd(INIT);
    simd1(TSTSTATE,STATE_COMMIT);
    simd1(CLRSTATE,STATE_COMMIT);
    simd_incr_any(-1,NFLAGS);
  }
  if (TSTDETAIL(10)) printf("<=== do_commitfin_lrf finish\n");
  cntc += clock-start;
}

build_full_cell_ints(cp,tok,flags,left,right,nleft,nright)
cellstate *cp,*nleft,*nright;
int tok,left,right,flags;
{
  cp->CELLTOK = tok;
  cp->CELLFLAGS = flags;
  if (left==NULL) cp->CELLLEFT = 0; else cp->CELLLEFT = left;
  if (right==NULL) cp->CELLRIGHT = 0; else cp->CELLRIGHT = right;
  if (nleft==NULL) cp->CELLNLEFT = 0; else cp->CELLNLEFT = nleft->self;
  if (nright==NULL) cp->CELLNRIGHT = 0; else cp->CELLNRIGHT = nright->self;
  cp->marks = 0;
}


/* ------------------------------------------------------------------- */
#define NUMB 1
#define TAK 2
#define TAKFAR 3
#define TAKCLOSE 4
#define TAKHOLD 5

#define PROBLEM_SIZE 3

/* ------------------------------------------------------------------- */
/* Build the initial term */
term_build(n)
int n;
{
  int i;
  cellstate *cp,*sp;
  cp = alloc_in(ROWNUM-3,4);
  root = cp->self;
  build_full_cell_ints(cp,TAKCLOSE,PROBLEM_SIZE,
                       3*PROBLEM_SIZE,2*PROBLEM_SIZE,NULL,NULL);
/*  build_full_cell_ints(cp,TAKCLOSE,3,10,6,NULL,NULL); */
}

/* ------------------------------------------------------------------- */
int reducedlim = 2;

/* ------------------------------------------------------------------- */
simd_execute()
{
  int start;
  progname = "tak";
  do_init();
  fetchvers=0; storevers=fetchvers;
  stat_init();

  while (1) { /* begin of main loop */
  stat_print_loop();
/* ------------------------------------------------------------------- */
  do_posn(0);
  /* TAK */
  do_posnl(2,"tak base");
  simd(INIT); SHOW;
  simd(ERASE); SHOW;
  simd1(CONST,NUMB); SHOW;
  simd2(EQ,ACC,TOK); SHOW;
  simd1(SETMARK,1); SHOW;

  if (CELLAT(root).marks&(1<<1)) break;

/* pull far numbers in close */
  simd(INIT); SHOW;
  simd1(CONST,TAKFAR); SHOW;
  simd2(EQ,ACC,TOK); SHOW;  /* am I a tak far away? */
  simd2(TSTMARKPTR,1,LEFT); SHOW;  /* and are my 3 kids #'s? */
  simd2(TSTMARKPTR,1,RIGHT); SHOW;
  simd2(TSTMARKPTR,1,FLAGS); SHOW;
  simd1(SETMARK,3); SHOW;
  simd1(CONST,TAKCLOSE); SHOW;
  simd2(MOVE,TOK,ACC); SHOW;  /* now I will be close */
  simd_incrptr(-1,LEFT); SHOW;
  simd_incrptr(-1,RIGHT); SHOW;
  simd_incrptr(-1,FLAGS); SHOW;
  simd_fetch(LEFT,LEFT,FLAGS); SHOW;
  simd(INIT); SHOW;                          /* must reinit */
  simd1(TSTMARK,3); SHOW;
  simd_fetch(RIGHT,RIGHT,FLAGS); SHOW;
  simd(INIT); SHOW;                          /* must reinit */
  simd1(TSTMARK,3); SHOW;
  simd_fetch(FLAGS,FLAGS,FLAGS); SHOW;

/* clock 31 */
  /* takclose stuff */
  simd(INIT); SHOW;
  simd1(CONST,TAKCLOSE); SHOW;
  simd2(EQ,ACC,TOK); SHOW;  /* am I a tak? */
  simd1(SETMARK,3); SHOW;
  simd2(LE,LEFT,RIGHT); SHOW; /* if x < y */
  simd1(CONST,NUMB); SHOW;
  simd2(MOVE,TOK,ACC); SHOW;
```

```
  simd2(MOVE,FLAGS,FLAGS); SHOW;  /* then z    ONLY DIFFERENCE */
  simd1(CLRMARK,3); SHOW;

  /* else */
  do_posn1(2,"tak gen");
  simd(INIT); SHOW;
  simd1(TSTMARK,3); SHOW;
  simd_alloc(NLEFT); SHOW;
  simd1(SETMARK,4); SHOW;
  simd_alloc(NRIGHT); SHOW;
  simd_alloc(NFLAGS); SHOW;
  simd1(SETMARK,5); SHOW;
  simd2(SETMARKPTR,6,NLEFT); SHOW;
  simd2(SETMARKPTR,6,NRIGHT); SHOW;
  simd2(SETMARKPTR,6,NFLAGS); SHOW;

  simd(INIT); SHOW; /* set token of decendents */
  simd1(TSTMARK,6); SHOW;
  simd1(CONST,TAKCLOSE); SHOW;
  simd2(MOVE,TOK,ACC); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd_store(NLEFT,RIGHT,RIGHT); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd_store(NLEFT,FLAGS,FLAGS); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd_store(NRIGHT,RIGHT,FLAGS); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd_store(NRIGHT,FLAGS,LEFT); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd_store(NFLAGS,RIGHT,LEFT); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd_store(NFLAGS,FLAGS,RIGHT); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd1(CONST,1); SHOW;
  simd2(SUB,LEFT,ACC);  SHOW;
  simd_store(NLEFT,LEFT,LEFT); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd2(SUB,RIGHT,ACC);  SHOW;
  simd_store(NRIGHT,LEFT,RIGHT); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,5); SHOW;  /* back at root */
  simd2(SUB,FLAGS,ACC);  SHOW;
  simd_store(NFLAGS,LEFT,FLAGS); SHOW;

  simd(INIT); SHOW;
  simd1(TSTMARK,4); SHOW;  /* back at root (even if fail) */
  simd1(CONST,TAKFAR); SHOW;

  simd2(MOVE,NTOK,ACC); SHOW;
  simd2(COMMITMARK,5); SHOW; /* commit */
/* clock = 134 on commit */

  do_posn(1); do_posn1(2,"special");

  /* excluding actions in autonomous processes from basic counts */
  do_stat_save();
  special = 1;

  do_commitfin_lrf();
  do_relocate();
  do_forward();                    /* do we need this? */
  do_delete();

  special = 0;
  do_stat_restore();

/* clock = 168 */
  mainloop++;
  } /* end of main loop */

  do_final();
}
```